Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Masterarbeit

Evaluierung und Erweiterung von MapReduce-Algorithmen zur Berechnung der transitiven Hülle ungerichteter Graphen für Entity Resolution Workflows

Leipzig, September 2013

vorgelegt von
Ziad Sehili
Master of Science Informatik

Betreuender Hochschullehrer:
Prof. Dr. Erhard Rahm
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Abstract

Im Bereich von *Entity-Resolution* oder *deduplication* werden aufgrund fehlender global eindeutiger Identifikatoren Match-Techniken verwendet, um zu bestimmen, ob verschiedene Datensätze dasselbe Realweltobjekt darstellen. Die inhärente quadratische Komplexität führt zu sehr langen Laufzeiten für große Datenmengen, was eine Parallelisierung dieses Prozesses erfordert. MapReduce ist wegen seiner Skalierbarkeit und Einsetzbarkeit in Cloud-Infrastrukturen eine gute Lösung zur Verbesserung der Laufzeit. Außerdem kann unter bestimmten Voraussetzungen die Qualität des Match-Ergebnisses durch die Berechnung der transitiven Hülle verbessert werden. Die Berechnung der transitiven Hülle eines Graphen ist von Natur aus ein iterativer Prozess. Ein naiver Ansatz berechnet sie *linear*, i.e. nach d Iterationen, wobei d die Tiefe des Graphen ist. In dieser Arbeit wird am Beispiel der Entity Resolution die Verwendung von MapReduce für die iterative und verteilte Berechnung der transitiven Hülle untersucht. Der vorgeschlagene Algorithmus Smart-MR operiert nur auf azyklischen Graphen und konvergiert nach genau $\log d$ Iterationen. Die drei weiteren Algorithmen Cyc-Smart-MR, Full-TC-MR und CC-MR arbeiten alle auf beliebigen ungerichteten Graphen und weisen ebenso ein logarithmisches Verhalten auf.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

1 Einleitung

Das aktuelle Trendthema im IT-Bereich ist Big Data, und das nicht ohne Grund. Das Datenvolumen im Internet verdoppelt sich alle zwei Jahre. Wurde es im Jahr 2011 auf 1,8 Zettabyte[1] geschätzt, so betrug es 2012 bereits 2,8 Zettabyte[2]. Die Daten werden vor allem von den sozialen Netzwerken, Sensoren und Protokoll- und Logdaten von Rechnern erzeugt. Auch das überall verfügbare Internet trägt zum rasanten Wachstum des Datenvolumens bei. Der Umgang mit Daten derartiger Dimensionen wirft zwei Fragen auf: *a)* Wie werden solche Datenmengen gespeichert? und *b)* Wie werden sie bearbeitet oder analysiert? Es ist offensichtlich, dass die Lösungen auf der Parallelisierung beruhen. Google begegnete dem Problem der Speicherung seiner gewaltigen Datenmenge mit *Google File System* (GFS)[GGL03], einem verteilten fehlertoleranten Dateisystem, das Dateien partitioniert und die resultierenden Blöcke auf mehrere Rechner repliziert. Für die Analyse und Verarbeitung dieser Daten entwickelte Google das MapReduce (MR) Framework [DG04]. Dabei handelt es sich um die Implementierung eines Programmiermodells, bestehend aus den zwei generischen Funktionen *map* und *reduce*, deren Implementierungen unter Ausnutzung von Datenparallelität gleichzeitig auf tausenden von Rechnern ausgeführt werden können. Hadoop [Had] ist eine frei verfügbare und sehr populäre Implementierung des MapReduce-Frameworks sowie des darunterliegenden verteilten Dateisystems *Hadoop Distributed File System* (HDFS). Die Verbreitung von Cloud-Infrastrukturen wie Amazon EC2, die das Mieten von Hardware on-demand per Mausklick erlauben, machen es zusätzlich sehr attraktiv, MapReduce-basierte Lösungsansätze für datenintensive Probleme zu entwickeln.
Entity Resolution (ER) ist ein wichtiger Teilbereich der Datenintegration, dessen Ziel die Entdeckung von Entitäten in einer oder mehreren Datenquellen ist, die sich auf dasselbe Realweltobjekt beziehen. In [Chr12] wurden unterschiedliche Fachgebiete erwähnt, die ER verwenden. Diese variieren von traditionellen Bereichen wie Volkszählung und Gesundheit bis hin zu neuen Themen wie Staatssicherheit und E-Commerce. All diese Bereiche werden meist

[1] http://germany.emc.com/leadership/programs/digital-universe.htm
[2] http://www.finanzen100.de/finanznachrichten/wirtschaft/2-8-zettabyte-2012-das-internet-datenvolumen-explodiert_H331434999_43714/

mit sehr großen Datenmengen konfrontiert, sodass skalierbare parallele Techniken benötigt werden. Dedoop (**De**duplication with Ha**doop**) [KTR12a] ist ein MapReduce-basiertes Tool zur Lösung von laufzeitintensiven ER-Problemen. Seine Weboberfläche ermöglicht u.a. die Definition der einzelnen Schritte einer ER-Strategie, wie die Blockbildung, die Ähnlichkeitsberechnung und die Klassifizierung. Diese werden anschließend als MapReduce-Programme in einem Hadoop-Cluster ausgeführt. Bei einem Match-Verfahren führt das Blocking in Verbindung mit restriktiven Klassifikationsregeln dazu, dass korrekte matches nicht gefunden werden. In solchen Fällen kann die Verwendung der transitiven Hülle (im Folgenden auch TC[1]) sinnvoll sein, um die Gesamtqualität des Match-Ergebnisses zu verbessern [KTR12b]. Dabei wird das finale Match-Ergebnis M um die Korrespondenz (a, c) erweitert, sofern $(a, b) \in M$ und $(b, c) \in M$ aber $(a, c) \notin M$.
Die vorliegende Arbeit beschäftigt sich mit der Umsetzung der transitiven Hülle mittels des Programmiermodells MapReduce für verschiedene Typen von Graphen. Basierend auf den Ergebnissen wird Dedoop um die Möglichkeit erweitert, am Ende eines Match-Workflows die TC zu berechnen. Meine Beiträge sind die Folgenden:

- Untersuchung der Probleme bei der Berechnung der TC mittels MapReduce und die Implementierung des Smart-MR Algorithmus, der die TC eines ungerichteten azyklischen Graphen in $\log d$ Iterationen berechnet,
- Implementierung drei weiterer Algorithmen (Cyc-Smart-MR, Full-TC-MR und CC-MR) zur Berechnung der TC beliebiger Graphen mit logarithmischem Verhalten,
- Erweiterung von Dedoop mit den drei letzten Algorithmen und Evaluierung ihrer Laufzeiten.

Diese Arbeit ist wie folgt strukturiert. Im folgenden zweiten Kapitel wird das Framework Hadoop mit seinem Programmiermodell MapReduce und seinem Dateisystem HDFS vorgestellt. Außerdem werden die Schritte, aus denen ein ER-Workflow besteht, präsentiert und die prinzipielle Umsetzung von ER-Workflows durch Dedoop erklärt. Im dritten Kapitel wird auf verwandte Arbeiten zur sequentiellen und parallelen Berechnung der TC eingegangen. Das vierte Kapitel präsentiert eine Methode, die die transitive Hülle mit MapReduce berechnet, und analysiert die dadurch entstehenden Probleme. Zur Behebung dieser Probleme werden mehrere entsprechende Algorithmen vorgestellt. Die Evaluation diser Algorithmen erfolgt dann im fünften Kapitel, bevor ich die Arbeit mit einer Zusammenfassung abschließe.

[1] angelehnt an die englische Übersetzung „Transitive Closure“

2 Grundlagen

Dieses Kapitel erläutert anhand des Frameworks Hadoop, wie große Datenmengen in einem verteilten Dateisystem gespeichert und mit MapReduce parallel analysiert werden können. Außerdem und bevor im letzten Abschnitt das Tool Dedoop präsentiert wird, welches das Programmiermodell MapReduce für die Abwicklung von komplexen Entity Resolution-Problemen verwendet, soll auf den Prozess von Entity Resolution eingegangen werden.

2.1 Hadoop

Die stetig wachsende Menge heterogener und zum Teil unsauberer Daten im Internet war der Grund für die Entwicklung neuer Lösungen, bei denen die Skalierbarkeit im Mittelpunkt steht. Traditionelle Datenbanksysteme waren trotz ihrer Bedeutung aber aufgrund ihrer mangelnden Skalierbarkeit und Fehlertoleranz sowie ihrer festen Schemata nicht vertretbar. Der Trend ging in Richtung „gängige Computer“ (commodity computers), die horizontal skalieren (tausende von Computern), deren Ausfall die Regel und keine Ausnahme ist. Darüber hinaus sind die Daten, die sie speichern oder bearbeiten, meistens groß und semi- oder unstrukturiert. Google löste das Problem der Speicherung seiner gewaltigen Datenmenge mit dem verteilten Dateisystem Google File System (GFS) und entwickelte das Programmiermodell MapReduce für die Analyse dieser Daten. Hadoop ist ein Open Source Framework, das MapReduce implementiert und ein verteiltes Dateisystem namens HDFS zur Verfügung stellt. Ein Hadoop-Cluster kann aus tausenden von Rechnern bestehen, die sowohl bei der verteilten Speicherung großer Daten als auch bei der parallelen Ausführung von MapReduce-Programmen beteiligt sein können. Einer der größten Vorteile dieser Kombination ist die Datenlokalität bei der Verarbeitung der Daten. Im Folgenden werden die zwei Kernkomponenten von Hadoop, HDFS und MapReduce, dargestellt und es wird erläutert, wie sie zur Datenlokalität beitragen.

2.1.1 Hadoop Distributed File System

In einem verteilten Dateisystem wird transparent auf Dateien und Ordner zugegriffen, die auf einem anderen, über ein Netzwerk erreichbaren Computer liegen. Obwohl die Grundidee dieselbe ist, unterscheiden sich die existierenden verteilten Dateisysteme in vielerlei Hinsicht, beispielsweise bezüglich der Architektur, der Konsistenz oder der Fehlertoleranz [TMC+08]. Die Unterschiede werden meistens vom Verwendungszweck bestimmt. Hadoop Distributed File System, abgekürzt HDFS, ist eins von diesen Dateisystemen, bei dem einige Designentscheidungen in den Vordergrund gestellt wurden, darunter:

Dateigröße: Es ist nicht selten, dass die Größe einer Datei in HDFS mehrere Terabyte erreicht. Da eine Datei von solcher Größe nicht auf einer einzelnen Maschine gespeichert werden kann, wird sie in Chunks oder Blöcke von meistens 64 MB partitioniert. Diese Blöcke werden auf verschiedenen Rechnern im Cluster gespeichert.

Datenzugriff und Konsistenz: HDFS wurde für batch-Aufgaben optimiert. Diese Aufgaben lesen die Daten in einen Datenstrom (Streaming) ein, i. e. eine Datei wird sequentiell von ihrem Anfang bis zu ihrem Ende gelesen. Dies geschieht, weil eine zu lesende Datei meistens sehr groß ist, sodass ein hoher Durchsatz bei einem sequentiellen Festplattenzugriff effizienter ist als eine geringe Latenz bei einem wahlfreien Zugriff. Außerdem wird die Konsistenz einer Datei dadurch bewahrt, dass sie nur einmal geschrieben wird und mehrfach gelesen werden kann (write-once-read-many).

Hardware und Fehlertoleranz: Ein Hadoop-Cluster kann aus tausenden von Rechnern bestehen. Diese Rechner sind sogenannte „Commodity Computers“, i. e. keine hochwertigen sondern normale „gängige“ Rechner. Die Qualität und v. a. die Anzahl dieser Rechner führen dazu, dass die Ausfallwahrscheinlichkeit eines oder mehrerer Rechner groß ist.

Replikation: Da der Ausfall eines Rechners relativ häufig vorkommt, was den Verlust von Daten verursachen kann, wird jeder Block einer partitionierten Datei auf mehreren Rechnern repliziert (in der Regel drei Replikationen pro Block). Wenn durch einen Ausfall die Anzahl der Replikationen eines Blockes sinkt, wird aus den verbleibenden Replikationen eine neue Kopie in einem anderen Rechner erzeugt.

Für die Verwirklichung dieser Anforderungen unterteilt ein HDFS-Cluster seine Rechner (im Folgenden auch Knoten genannt) in zwei Arten, *Namenode* und *Datanode(s)*. Beide Arten sind in Wirklichkeit nur *Daemons*, welche auf einem Host-Computer, meistens Linux, laufen. Die Datanodes sind die Knoten,

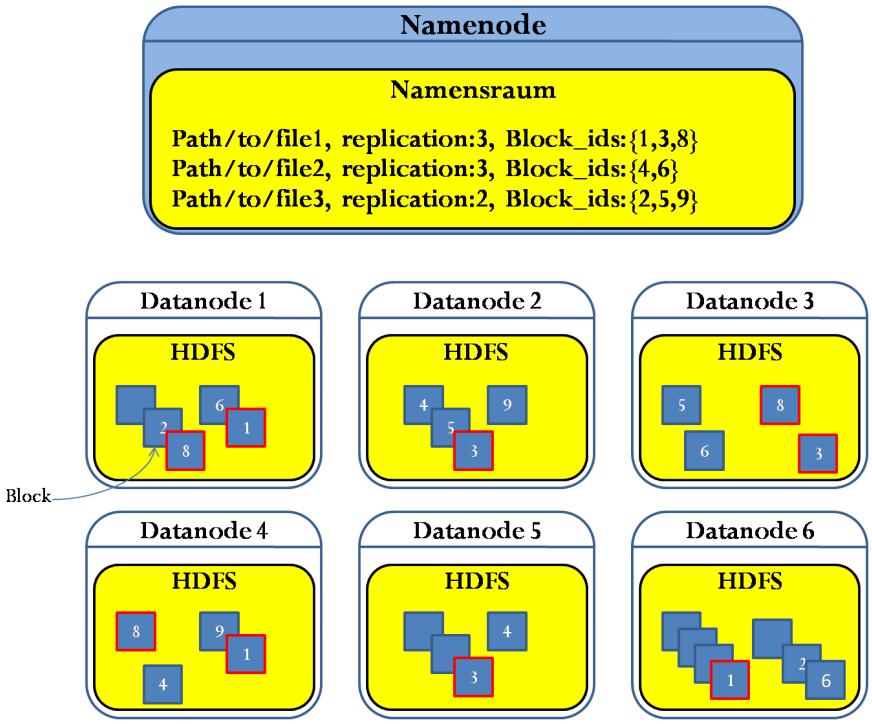

Abbildung 2.1: Beispiel von den Informationen eines Namensraums

die die Blöcke speichern. Sie sind in ständiger Kommunikation mit dem Namenode und schicken ihm einen Bericht, über welche Blöcke sie verfügen und über die Anzahl der Kopien jedes Blockes. Der Namenode, ein einziger Knoten im Cluster, ist für die Verwaltung vom Namensraum und von den Metadaten der Dateien zuständig. Da eine Datei in HDFS aus mehreren Blöcken besteht, wird im Namensraum nicht nur die hierarchische Struktur der Datei sondern auch die Anzahl ihrer Replikationen und die Ids ihrer Blöcke gespeichert. Die Abbildung 2.1 zeigt die Art der Daten, die von den jeweiligen Knoten eines HDFS-Clusters verwaltet werden. Der Namenode speichert z. B. neben der hierarchischen Struktur der Datei *file1* die Ids der Blöcke, aus denen sie besteht $\{1, 3, 8\}$ und die Anzahl der Replikationen dieser Blöcke, hier drei. Diese werden in den Datanodes 1 bis 6 physisch gespeichert und repliziert. Im Falle des Ausfalls von *Datanode 5* merkt der Namenode anhand der von den Datanodes geschickten Berichte, dass die Anzahl der Replikationen des Blockes mit der Id 3 unter dem vorgegebenen Replikationswert liegt. Anschließend weist er einen Datanode an, diesen Block aus dem Datanode 2 oder 3 wiederherzustellen. Das Mapping zwischen Block-Ids im Namensraum und ihren Speicherorten in den Datanodes wird nicht persistent beim Namenode gespeichert, sondern erst beim Start der Datanodes durch die von ihnen geschickten Berichte an den Namenode ermittelt. Zusätzlich verwaltet

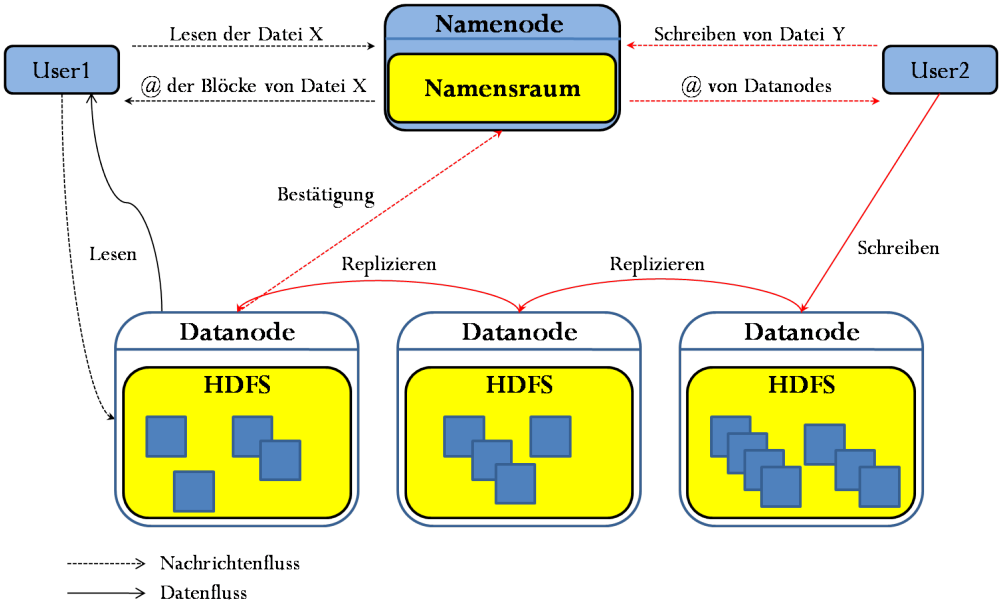

Abbildung 2.2: Lesen und Schreiben in HDFS

der Namenode die Log-Daten der Änderungen, die an den Daten durchgeführt wurden.

Lesen und Schreiben in HDFS Diese zwei Prozesse wurden so konzipiert, dass der Namenode im Wesentlichen entlastet und nur bei der Verwaltung und Übermittlung der Metadaten beteiligt wird. Die Übertragung der Daten erfolgt direkt zwischen dem User und den Datanodes. Wenn ein User eine Datei lesen möchte, kontaktiert er zunächst den Namenode und fragt nach den Datanodes, die die Blöcke der betreffenden Datei haben. Der Namenode antwortet mit den entsprechenden Adressen. Mit diesen liest der User die Blöcke in der richtigen Reihenfolge direkt aus den Datanodes, ohne weitere Beteiligung des Namenodes. Dieser Prozess wird in Abbildung 2.2 mit dem *User1* und den schwarzen Pfeilen dargestellt. Die roten Pfeile zeigen die Schritte einer Schreibanfrage vom *User2*. Beim Schreiben kontaktiert der User ebenfalls den Namenode mit einer Schreibanfrage. Der Namenode protokolliert dies zunächst in die Logdatei, dann antwortet er mit einer Liste von Adressen von Datanodes, die diese Datei aufnehmen werden. Der User schickt seine Datei und die Adressliste an einen Datanode, der die Datei lokal speichert und zwecks Replikation zum nächsten Datanode auf der Adressliste weiterleitet. Wenn der letzte Datanode mit dem Schreiben fertig ist, informiert er den Namenode, dass die Datei erfolgreich geschrieben und repliziert wurde. Erst dann kann der Namenode den Namensraum mit dieser Datei und ihren Replikationen aktualisieren.

2.1.2 MapReduce

Mit paralleler Datenverarbeitung wird durch den Einsatz von mehreren Prozessoren versucht, bessere Performance zu erzielen. Ein wichtiges Konzept dabei ist die Datenlokalität. Es ist offensichtlich, dass für bestimmte Aufgaben jeder von den Prozessoren *nicht* die gesamten Daten benötigt, sondern nur einen Teil davon, der möglichst lokal ist. Daher ist es besser, die Aufgaben (oder den Quellcode) an jene Rechner zu schicken, die eine Partition der Daten haben und diese lokal bearbeiten können, i. e. es findet kein Datenaustausch zwischen den Rechnern statt. Das ist das Prinzip der Parallelisierung in MapReduce.

MapReduce wurde als Programmiermodell für die Analyse von großen Datenmengen bei Google entwickelt. Dabei handelt es sich um zwei generische Funktionen, *map* und *reduce*, die vom User implementiert und in einem Cluster von mehreren Rechnern parallel ausgeführt werden. Die Daten sind am Anfang partitioniert und repliziert im Cluster. Jeder Rechner liest die lokalen Daten zeilenweise als $(key_{in}, value_{in})$-Paare und übergibt sie dem map-Task (map-Funktion in einem bestimmten Rechner). Der Key dieses Paares ist meistens der Byte-Offset einer Zeile und der Value ist die Zeile selbst. Die Ausführung der map-Funktion auf jedes gelesene Paar erzeugt ein anderes temporäres $(key_{tmp}, value_{tmp})$-Paar[2]. Außerdem sortiert der map-Task seine generierten Paare nach Key und speichert sie temporär lokal. Es folgt eine Umverteilung der kompletten Ausgaben der map-Phase, indem die temporären Paare mit demselben Key aus allen map-Tasks zum selben reduce-Task gesendet werden. Die reduce-Tasks gruppieren ihre erhaltenen Paare nach Keys in der Form $(key_{tmp}, List[value_{tmp}])$-Paar. Auf diese Paare wird die reduce-Funktion ausgeführt, die schließlich ein aggregiertes Paar $(key_{final}, value_{final})$ ausgibt. Der Prozess kann anhand der Abbildung 2.3 erläutert werden. Es handelt sich um das Zählen des Auftretens von Buchstaben in einer Datei. Die Daten sind auf zwei Knoten gespeichert. Jeder Knoten beinhaltet einen map-Task, der die Daten zeilenweise als $(key_{in}, value_{in})$ liest. Zum Beispiel liest der map-Task$_1$ das Paar $(0, ABBA)$, wobei 0 der Byte-Offset und $ABBA$ die Daten der Zeile sind. Er generiert für jeden gelesenen Buchstaben ein Paar mit dem Buchstaben als Key und dem Wert 1 als Value, i.e. für die Zeile $ABBA$ generiert der map-Task$_1$ die Paare $(A, 1)$, $(B, 1)$, $(B, 1)$ und $(A, 1)$. Dasselbe geschieht mit den anderen Zeilen und mit map-Task$_2$. Die generierten Paare von map-Task$_1$ und map-Task$_2$ werden dann auf die reduce-Tasks 1 und 2 partitioniert, indem alle Paare mit demselben Key beim selben reduce-Task gruppiert werden. Der reduce-Task$_2$ erhält z. B. alle Paare mit dem Key B und gruppiert sie in das neue Paar $(B, [1, 1, 1, 1])$. Abschließend summiert die reduce-Funktion die Werte der

[2] Es ist möglich, dass eine map- oder reduce-Funktion mehr als ein einziges key-value-Paar oder gar keins ausgibt.

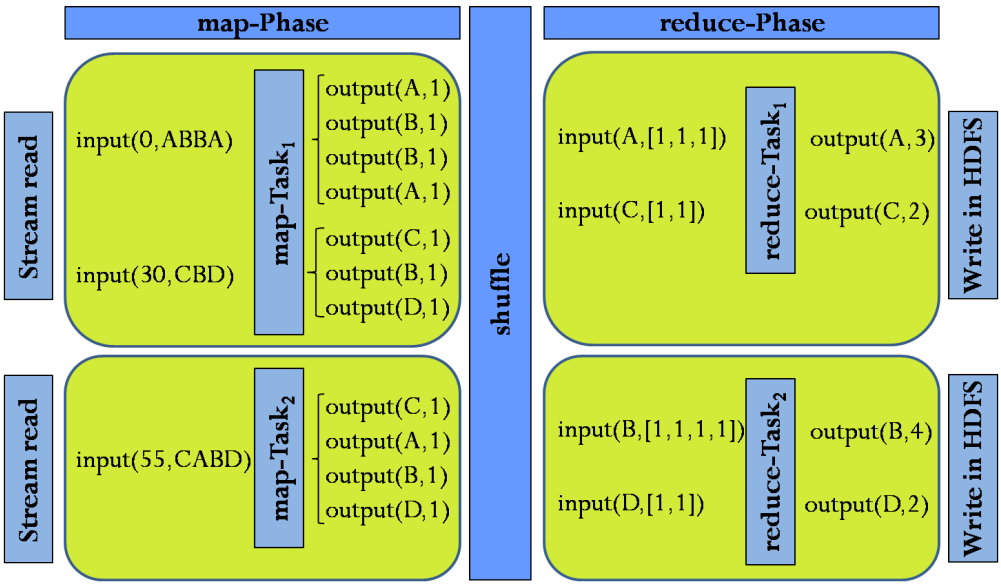

Abbildung 2.3: Lösungsansatz für das Problem *Word Count* in MapReduce

Values jedes Keys und gibt das Paar (Buchstabe, Summe) als key-value-Paar aus. Beispielsweise gibt die reduce-Funktion aus dem Paar $(B, [1, 1, 1, 1])$ das finale Paar $(B, 4)$ aus, das die Anzahl des Buchstabens B in der Datei darstellt.

Job Ausführung in Hadoop Im Hadoop Framework werden Aufgaben oder Programme, die in MapReduce implementiert sind, als Jobs bezeichnet. Die Daten, die der Job bearbeitet, liegen bereits in HDFS als partitionierte replizierte Blöcke von meistens 64 MB, wo sie von nun an als Splits betrachtet werden. Ein Split hat oft die Größe eines Blocks, kann aber bei Bedarf größer oder kleiner gesetzt werden. Bei MapReduce, genau wie bei HDFS, werden zwei Arten von Knoten oder eher Daemons unterschieden, a) der JobTracker, der für die Koordinierung der gesamten Ausführung des Jobs zuständig ist, und b) die TaskTrackers, die die einzelnen map- und reduce-Funktionen ausführen. Die verschiedenen Schritte der Ausführung eines MapReduce-Jobs werden mittels der Abbildung 2.4 erklärt. Hier wird der Einfachheit halber angenommen, dass nur ein Job ausgeführt wird. Wenn der Client den Job zur Ausführung schickt, wird der JobTracker als erster kontaktiert, um eine Job-Id von ihm zu bekommen (Schritte 1 und 2). Dann überprüft das Framework, dass das Verzeichnis für die Ausgabe nicht existiert, und ermittelt die Ids der betreffenden Splits (Schritt 3). Anschließend kopiert es die JARs (MapReduce Programme), die Ids der Splits und die Konfigurationsdatei des Jobs in HDFS mit hoher Replikationsrate und informiert den JobTracker, dass der Job bereit zur Ausführung ist (4 und 5). Der JobTracker extrahiert die Ids der Splits aus HDFS und erstellt

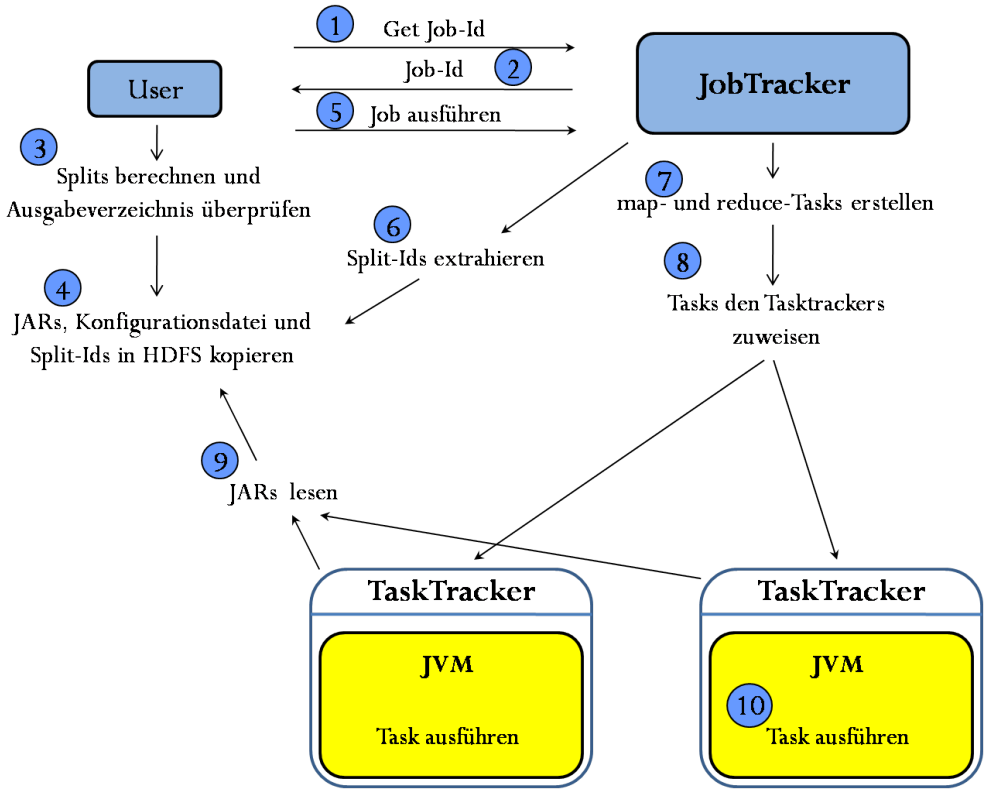

Abbildung 2.4: Schritte der Ausführung eines MapReduce-Jobs in Hadoop

für jedes Split einen map-Task und so viel reduce-Tasks, wie der User es in der Konfigurationsdatei vorgegeben hat (6 und 7). Nach der Erstellung der map- und reduce-Tasks erfolgt die Zuweisung dieser Tasks zu den Knoten. Für die Gewährleistung der Datenlokalität bekommen jene TaskTrackers, welche Blöcke besitzen, deren Ids den Ids von den Splits entsprechen, die Anweisung vom JobTracker, die map-Funktion auf ihren Daten auszuführen. Daraufhin holen sich diese TaskTrackers die JAR-Dateien aus HDFS, starten eine neue Java virtuelle Maschine (JVM) und führen darin ihre Quellcodes aus. Außerdem benachrichtigen sie den JobTracker über den Fortschritt ihrer Ausführung (Schritte 8, 9 und 10). Im Falle der reduce-Tasks kann das Framework nicht wirklich von der Datenlokalität profitieren, da die Daten umverteilt wurden. Deswegen bekommt der erste frei gewordene TaskTracker die Anweisung, die reduce-Funktion auszuführen. Stellt der JobTracker fest, dass alle Tasks erfolgreich durchgeführt wurden, benachrichtigt er den Client über den Erfolg der Ausführung seines Jobs.

2.2 Dedoop und Entity Resulotion

2.2.1 Entity Resolution

Um zwei Datenbanken zu fusionieren oder um Produktpreise in einem Portal zu vergleichen, ist es wünschenswert, wenn nicht sogar notwendig, Duplikate zu erkennen und zu entfernen. Entity Resolution (ER) ist ein wichtiger Teilbereich der Datenintegration, dessen Ziel die Entdeckung von Entitäten ist, die aus einer oder mehreren Datenquellen stammen und die sich auf dasselbe Realweltobjekt beziehen. Die Größe und Heterogenität der Daten sowie die Komplexität des Problems führen dazu, dass der Prozess von Entity Resolution als Workflow in mehreren Schritten durchgeführt wird. Diese definieren die gesamte ER-Strategie. In [Chr12] wurden die in Abbildung 2.5 dargestellten Schritte erkannt:

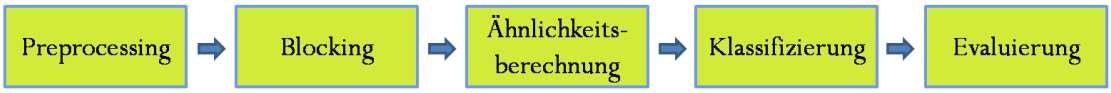

Abbildung 2.5: Die Schritte eines Entity-Resolution-Workflows

a) Preprocessing: Da die Daten meistens aus verschiedenen Quellen stammen, können sie sehr heterogen sein. In der Preprocessing Phase wird versucht, die Daten zu säubern und in ein möglichst einheitliches Schema zu überführen. Je nach Anwendung können z. B. Stopp-Wörter entfernt, Abkürzungen durch Wörter ersetzt oder auch komplexe Attribute in kleinere Einheiten zerlegt werden.

b) Indexierung oder Blockbildung: Die Kernaufgabe beim Matching besteht darin, Datensätze miteinander zu vergleichen. Intuitiv würde man jede Entität mit allen anderen vergleichen. Für Datenbanken und Dateien, deren Größe im Tera- und Petabyte-Bereich liegt, ist solch ein naiver Ansatz ineffizient und nicht akzeptabel, und das aus zweierlei Gründen. Einerseits steigt die Anzahl der Vergleiche quadratisch mit wachsender Datenmenge. Andererseits werden die meisten Vergleiche zwischen Entitäten durchgeführt, die eindeutig kein match darstellen. Um unnötige Vergleiche zu vermeiden, werden die Datensätze durch ein einfaches Kriterium in Blöcke geclustert. Das Kriterium kann z. B. ein Attribut sein. Häufig wird aber ein neues zusammengesetztes Attribut, ein sogenannter *Blocking key*, berechnet. Beispielsweise können die drei ersten Buchstaben mancher Attribute konkateniert werden. In beiden Fällen gehören alle Entitäten, die denselben Blocking-key besitzen, zum selben Block. Die Vergleiche erfolgen dann nur zwischen Entitäten desselben Blocks.

c) Vergleich von Entitäten: Mit einem *einfachen Kriterium* wurden *wahrscheinlich ähnliche* Datensätze in Blöcke geclustert. Jetzt müssen diese Datensätze auf der Attributsebene miteinander verglichen werden. Je mehr ähnliche Attribute zwei Entitäten besitzen, umso wahrscheinlicher ist es, dass sie dasselbe Realweltobjekt darstellen. Der Grad der Ähnlichkeit zweier Attribute wird meistens durch einen numerischen Wert zwischen 0 und 1 dargestellt. Ein Wert näher zu 1 deutet auf eine große Ähnlichkeit hin. Es existieren viele Funktionen, die den Ähnlichkeitsgrad zwischen zwei Attributen ermitteln. Die meisten von ihnen sind für Zeichenketten gedacht, z.B. *Levenshtein edit distance* oder *n-gram based string comparison*. Methoden für andere Datentypen existieren auch [ES07]. Das Ergebnis des Vergleiches zweier Entitäten ist ein Vektor, dessen Länge der Anzahl der miteinander verglichenen Attribute entspricht. Die Werte des Vektors, alle zwischen 0 und 1, repräsentieren den Ähnlichkeitsgrad dieser Attribute.

d) Klassifizierung: Anhand des Ähnlichkeitsvektors werden die Entitäten-Paare in zwei Gruppen geteilt, match und nicht-match. Es existieren verschiedene Methoden für die Verwendung des Vektors bei der Klassifizierung[ES07]. Die einfachste von ihnen ist die Aggregation der Werte des Vektors, z. B. die absolute oder normalisierte Summe seiner Werte. Alle aggregierten Werte, die oberhalb eines vordefinierten Schwellenwerts liegen, werden als match betrachtet, die anderen nicht.

e) Evaluierung des Match-Ergebnisses: Die abschließende Phase stellt die größte Herausforderung beim Data-Matching-Prozess dar. Ein Match-Verfahren wird u.a. durch zwei Eigenschaften beurteilt, *Genauigkeit* und *Vollständigkeit*, die im nächsten Kapitel näher betrachtet werden. Eine komplette Überprüfung des Match-Ergebnisses durch Menschen auf seine Genauigkeit und Vollständigkeit ist aus zwei Gründen nicht realisierbar. Erstens können Menschen manche Entitäten-Paare nicht eindeutig als match oder nicht-match klassifizieren, zweitens ist die zu überprüfende Datenmenge zu groß. Eine Lösung dafür wäre die Anwendung des Match-Verfahrens auf *gold standard*. In diesem Kontext ist *gold standard* eine Menge von Entitäten-Paaren, die schon als match und nicht-match annotiert sind. Anschließend werden Genauigkeit und Vollständigkeit des Ergebnisses durch den Vergleich mit den annotierten Daten berechnet. Das Problem bei diesem Ansatz sind die fehlenden *gold standards*.

2.2.2 Dedoop

Aufgrund seiner Art der Parallelisierung eignet sich das Programmiermodell MapReduce gut für das Entity-Resolution-Problem. Abbildung 2.6 zeigt ein Beispiel für die Verwendung von MapReduce zur Lösung eines einfachen ER-Problems. Zwei map-Prozesse lesen die Daten aus dem verteilten Dateisystem

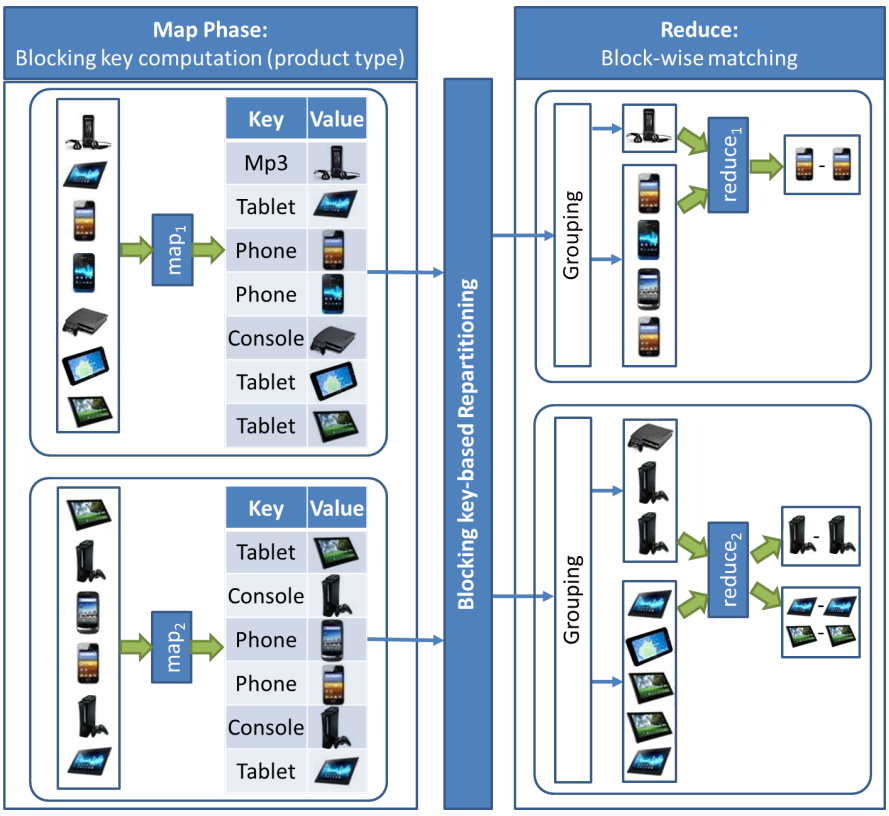

Abbildung 2.6: Deduplizierung von Produktdatensätzen mit MR. [[KR13]]

und wenden die map-Funktion an, in der eine Blockschlüsselberechnung (hier Produkttyp) erfolgt. Die Entitäten werden dann dynamisch auf zwei reduce-Prozesse umverteilt. Alle Entitäten desselben Produkttyps werden zum selben reduce-Prozess geschickt. Die reduce-Funktion wird für jeden Produkttyp aufgerufen. Innerhalb von reduce erfolgt die paarweise Ähnlichkeitsberechnung der Entitäten. Obwohl sich solche Match-Aufgaben leicht in MapReduce umsetzen lassen, ist die Spezifikation raffinierter Lösungen für bestimmte Probleme in diesem Programmiermodell und ihre Ausführung in verschiedenen Clustern recht kompliziert. Außerdem kann eine Änderung der Strategie in der Blockbildung oder in der Berechnung der Ähnlichkeit zu redundanter Arbeit führen. Dedoop, **De**duplication with Ha**doop**, ist ein auf MapReduce basierendes Tool für die Definition komplexer ER-Workflows. Der User kann mit Dedoop seine ER-Strategie definieren und in einem Hadoop-Cluster ausführen, ohne dabei über MapReduce-Kentnisse verfügen zu müssen.

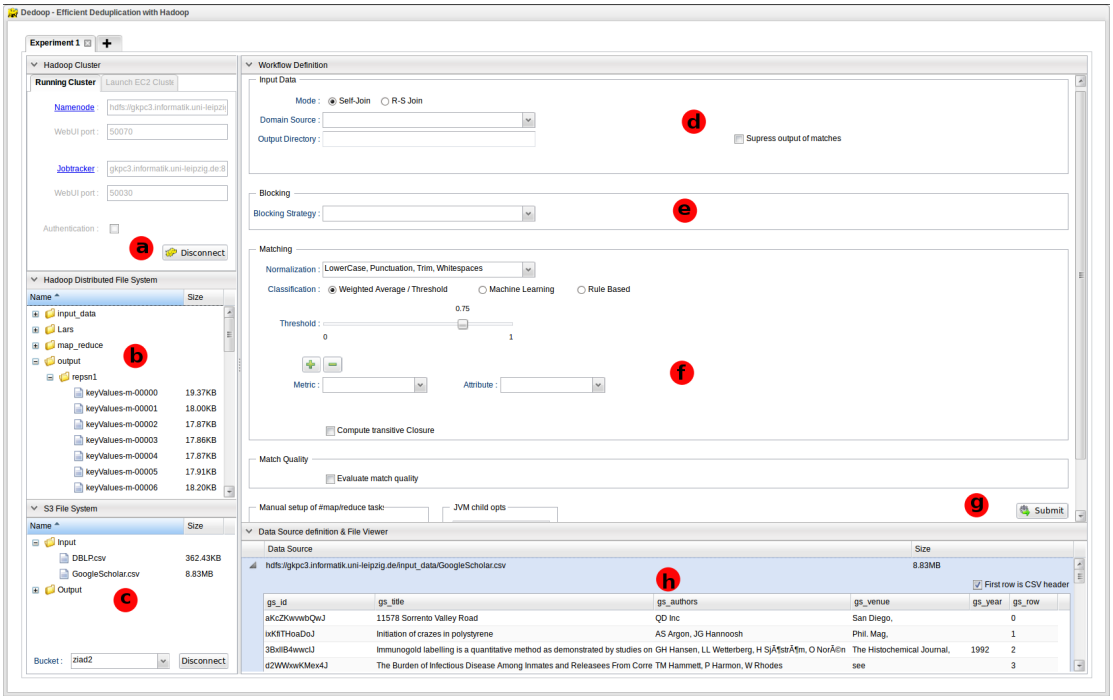

Abbildung 2.7: Dedoop-Oberfläche

2.2.2.1 Dedoop-Oberfläche

Für die Ausführung eines ER-Workflow stellt Dedoop dem User eine vielfältige Weboberfläche zur Verfügung. Sie ist im Wesentlichen in zwei Teile unterteilt. Der erste Teil ermöglicht die Herstellung einer Verbindung mit einem Hadoop-Cluster und die Verwaltung verschiedener Dateisysteme. Der zweite Teil dient als Oberfläche für die Definition einer ER-Strategie. Anhand der Abbildung 2.7 werden die Funktionen der einzelnen Komponenten dieser Oberfläche erläutert:

- **Herstellung der Verbindung:** Der User kann sich mit einem oder mehreren Hadoop-Clusters verbinden (Abbildung 2.7.a). Für die Verbindung mit einem lokalen Hadoop-Cluster benötigt er lediglich die Adressen des Namenodes und des JobTrackers. Noch einfacher läuft die Verbindung mit einem Hadoop-Cluster auf Amazon EC2. Dedoop stellt dafür eine Oberfläche zur Eingabe der verschiedenen Parameter des Hadoop-Clusters zur Verfügung, z. B. die Zugangs- und Geheimschlüssel, die Anzahl und den Typ der Knoten oder auch die Anzahl der Tasks pro Knoten.

- **Darstellung und Verwaltung der Daten:** Nach einer erfolgreichen Verbindung mit einem Hadoop-Cluster wird sein HDFS-Dateisystem in einer baumähnlichen Struktur dargestellt (Abbildung 2.7.b). Hier hat der User die Möglichkeit, verschiedene gewöhnliche Aufgaben durchzuführen, wie Dateien in das lokale Dateisystem hoch- bzw. aus ihm herunterladen, Dateien löschen oder umbenennen.
- **S3-Dateisystem:** Die Dateien oder genauer die Objekte, die im Amazon Cloud-Speichersystem S3 gespeichert sind, können ebenfalls als baumähnliches Dateisystem dargestellt werden (Abbildung 2.7.c).
- **Datenaustausch:** Durch die *drag-and-drop*-Funktion kann der User die Dateien zwischen verschiedenen Dateisystemen (lokal, HDFS, S3) austauschen.
- **Betrachtung von CSV-Dateien:** Dateien im CSV-Format können durch die *drag-and-drop*-Funktion in ein bestimmtes Feld (Abbildung 2.7.h) geschoben und dort angeschaut werden.
- **Definition eines ER-Workflows:** Die Felder in Abbildung 2.7.d, e und f vereinfachen die Definition der einzelnen Schritte einer ER-Strategie, beginnend mit den Eingabedateien bis hin zum Klassifikator.
- **Ausführung eines ER-Workflows:** Nach der Definition einer ER-Strategie kann der User sie zur Ausführung in ein Hadoop-Cluster schicken (Abbildung 2.7.g), wobei die einzelnen Schritte zu MapReduce-Jobs transformiert werden.

2.2.2.2 Arbeitsweise von Dedoop

Ein generelles ER-Problem und seine Transformation in ein MapReduce-Workflow in Dedoop ist in Abbildung 2.8 dargestellt. Die obere Schicht bildet den logischen ER-Prozess mit seinen Teilschritten ab, i. e. Blockbildung, Ähnlichkeitsberechnung und Klassifizierung. Ein weiterer erster Schritt, *Machine Learning*, wird nur dann durchgeführt, wenn die Klassifizierung auf dem maschinellen Lernen beruht. Ein Klassifikator wird in diesem Schritt trainiert und später für die Klassifizierung der Daten benutzt. Die untere Schicht stellt das entsprechende MapReduce-Workflow dar, das Dedoop zur Ausführung in einem Hadoop-Cluster generiert. Je nach Operationen, die vom User definiert wurden, können bis zu drei MapReduce-Jobs nacheinander gestartet werden, um ein ER-Workflow abzubilden. Im einfachsten Fall wird nur ein Job ausgeführt, der dem in Abbildung 2.6 gezeigten Beispiel entspricht, i. e. die Blockschlüssel werden in der map-Phase generiert und in der reduce-Phase

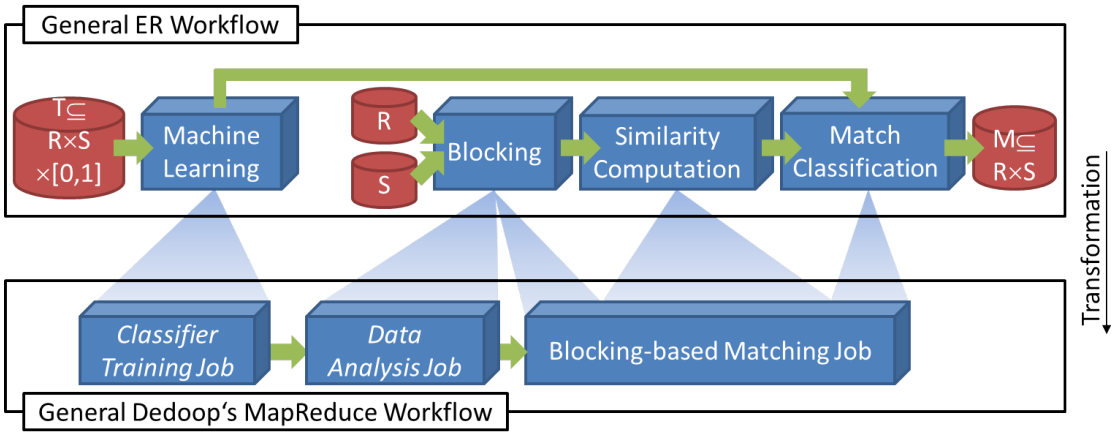

Abbildung 2.8: Realisierung von ER-Workflows mit MR.[[KR13]]

desselben Jobs werden die Entitäten nach Blockschlüsseln gruppiert und paarweise verglichen. Üblicherweise werden jedoch zwei weitere MapReduce-Jobs benötigt, um ein ER-Workflow durchzuführen. In dem ersten wird der Klassifikator trainiert, wenn die Klassifizierung eine Maschinelles-Lernen-Strategie verwendet. Eine wesentliche Bedeutung kommt dem zweiten MapReduce-Job, dem Data Analysis Job, zu, welcher Statistiken über die Datenverteilung und Blockgrößen sammelt. Diese werden zur Lastbalancierung und einer gleichmäßigen Ausnutzung der Cluster-Ressourcen verwendet. Eine ER-Strategie wird in Dedoop wie folgt definiert:

- Als erstes definiert der User die Eingaberelationen.
- Um den Suchraum zu reduzieren, wird im zweiten Schritt eine Blocking-Strategie definiert. Dedoop besitzt schon MapReduce-Implementierungen von verschiedenen Blocking-Strategien, z. B. das Standard Blocking oder Sorted Neighborhood.
- Für die Berechnung der Ähnlichkeit zwischen zwei Entitäten implementiert Dedoop auch mehrere Funktionen, die die Attribute der Entitäten vergleichen und einen Ähnlichkeitswert liefern. Ihre Ergebnisse werden bei der Klassifizierung der Entitäten in match und nicht-match benutzt.
- Der letzte Schritt ist das Abschicken der definierten Strategie, die Dedoop als MapReduce-Jobs zu einem Hadoop-Cluster zur Ausführung sendet. Während der Ausführung bleibt Dedoop in Kontakt mit dem Hadoop-Cluster und erhält ständig Informationen über den Zustand der Ausführung.

Die Definition eines ER-Workflows in Dedoop kann um die Berechnung der transitiven Hülle erweitert werden. Diese Erweiterung erfolgt unter

bestimmten Voraussetzungen und ermöglicht nicht nur die Verbesserung der Qualität des Match-Verfahrens, sondern auch die Überführung des Match-Ergebnisses in einen konsistenten Zustand. Das nächste Kapitel erläutert diese Idee und präsentiert unterschiedliche Algorithmen zur Berechnung der transitiven Hülle.

3 Die Transitive Hülle

Die transitive Hülle spielt nicht nur in der Mathematik eine bedeutende Rolle, sondern auch in Bereichen wie Graphentheorie, Datenbanken oder eben Entity Resolution. Zahlreiche Algorithmen wurden für ihre Berechnung entwickelt. Sie variieren von traditionellen sequentiellen Algorithmen bis hin zu parallelen Algorithmen mit logarithmischem Verhalten.

3.1 Definition der transitiven Hülle

Die transitive Hülle wird in verschiedenen Bereichen unterschiedlich definiert, jedoch mit derselben Semantik. Die hervorgehobene Idee in diesen Definitionen ist die Eigenschaft der Transitivität oder Erreichbarkeit von Knoten in einem Graphen. Drei Definitionen werden hier präsentiert:

3.1.1 Mathematische Definition

Sei eine beliebige Menge $S = \{x_1, x_2, ..., x_n\}$ und $R \subseteq S \times S$ eine binäre Relation über die Menge S. Die transitive Hülle von R ist die binäre Relation $R^* \subseteq S \times S$ mit den folgenden Eigenschaften:

1. $R \subseteq R^*$, i. e. wenn $x_i \, R \, x_j$, dann $x_i \, R^* \, x_j$.
2. R^* ist eine transitive Relation, i. e. für $x_i, x_j, x_z \in S$ wenn $x_i \, R^* \, x_j$ und $x_j \, R^* \, x_z$, dann $x_i \, R^* \, x_z$.

3.1.2 Algebraische Definition

Sei eine beliebige Menge $S = \{x_1, x_2, ..., x_n\}$ und $R \subseteq S \times S$ eine binäre Relation über die Menge S. Die transitive Hülle von R ist die Union von ihren *power*

Relationen R^i:

$$R^* = \bigcup_{i=1,2..n} R^i$$

wobei die *power* Relation R^n durch die Komposition von Relationen definiert wird:

$$R^n = \begin{cases} R & \text{if } n = 1 \\ R \circ R^{n-1} & \text{if } n > 1 \end{cases}$$

3.1.3 Graph-basierte Definition

Sei eine beliebige Menge $S = \{x_1, x_2, ..., x_n\}$ und $R \subseteq S \times S$ eine binäre Relation über die Menge S. Es wird ein Graph $G(V, E)$ definiert, mit $v_i \in V$ als Menge der Knoten dieses Graphen und $e(v_i, v_j) \in E$ als Menge seiner Kanten. Die Darstellung der Menge S und die Relation R durch den Graphen G ist wie folgt definiert:

1. Wenn $x_i \in S$, dann $x_i \in V$.
2. Wenn $x_i \, R \, x_j$, dann ist $e(x_i, x_j) \in E$ eine Kante im Graphen G.

Die transitive Hülle des Graphen G ist ein anderer Graph $G^*(V, E^*)$ mit der folgenden Eigenschaft: Eine Kante $e(x_i, x_j) \in E^*$ existiert, wenn es einen *Weg* von x_i nach x_j in G gibt.

Im verbleibenden Teil der Arbeit werden die Begriffe Relation und Graph abwechselnd verwendet und haben gemäß der letzten Definitionen dieselbe Bedeutung. Der folgende Abschnitt verdeutlicht diese Idee anhand von Beispielen und präsentiert die Semantik der transitiven Hülle in zwei Bereichen, Datenbank und Entity Resolution.

3.2 Einsatzmöglichkeiten der transitiven Hülle

3.2.1 Die transitive Hülle in den Datenbanken

Die Idee der transitiven Hülle wurde ursprünglich in deduktiven Datenbanken wie Datalog verwendet, um rekursive Anfragen zu beantworten. Betrachten wir das folgende Beispiel:

Beispiel 3.1:

$$erreichbar(x, y) :- \ fahrt(x, y) \tag{3.1}$$
$$erreichbar(x, y) :- \ fahrt(x, z) \ \ and \ \ erreichbar(z, y) \tag{3.2}$$

Das Datalog-Programm besteht aus zwei Regeln und behandelt das Problem der Erreichbarkeit. Die erste Regel (3.1) besagt, dass eine Stadt y von einer anderen Stadt x erreichbar ist, wenn es eine *direkte Fahrt* von x nach y gibt. Die zweite Regel (3.2) ist rekursiv und beschreibt die Erreichbarkeit wie folgt: Eine Stadt y ist von einer anderen Stadt x erreichbar, wenn es eine direkte Fahrt von x in irgendeine andere Stadt z gibt und y von z erreichbar ist.
Die Bedeutung von deduktiven Datenbanken und die Richtigkeit der Antworten auf Anfragen wird durch eine Semantik bestimmt. Drei Semantiken wurden dafür definiert [GM89], die *deklarative*, die *prozedurale* und die *Fixed-Point*-Semantik. In dieser Arbeit wird aufgrund ihrer Bedeutung und Verwendung in realen Datenbanksystemen lediglich die letzte Semantik, *Fixed Point*, für die Bearbeitung rekursiver Datalog-Programme erläutert.

Least Fixed Point Semantik: Mit dieser Semantik wird ein Herbrand-Modell schrittweise berechnet. Am Anfang wird eine Start-Menge, die das Ergebnis enthalten soll, als leer definiert. In jeder Iteration werden die Regeln des Datalog-Programms verwendet, um neue Fakten zu finden, z. B. wenn es eine Regel in der Form $A :- \ A_1, A_2, ...A_i$ gibt und $A_1, A_2, ...A_i$ schon in der Start-Menge enthalten sind, wird A zu dieser Menge hinzugefügt. Dieser Prozess wird fortgesetzt, bis keine neuen Fakten entdeckt werden, i. e. das *Least Fixed Point* wurde erreicht. Diese Vorgehensweise wird am besten anhand des Beispiels 3.1, das die Erreichbarkeit von Städten berechnet, verdeutlicht. Das Prädikat *erreichbar* wird mit einer leeren Menge Z repräsentiert. Da das Prädikat *fahrt* Fakten darstellt, wird die erste Regel (3.1) ausgeführt und deren Fakten werden zur Menge Z addiert. Die zweite Regel (3.2) verwendet einerseits die Fakten des Prädikats *fahrt*, andererseits das in Menge Z gespeicherte Ergebnis. Ihre Ausführung erzeugt Fakten, die ebenfalls zur Menge Z addiert werden. In der Iteration i werden Ergebnisse oder Fakten generiert, die in der nächsten Iteration $i + 1$ für die Ausführung der zweiten Regel (3.2) verwendet werden. Dieser Prozess wird so lange fortgesetzt, bis die zweite Regel keine neuen Ergebnisse mehr liefert.
Die relationalen Datenbanken können als deduktive Datenbanken betrachtet werden [CGT89, UGMW01]. Eine Relation kann als Prädikat und ihre Tupel als Fakten (*ground facts*) angesehen werden. Diese Relationen bzw. Prädikate werden in zwei Kategorien unterteilt, die *Extentional Database (EDB)* und die *Intensional Database (IDB)*. Bei *Extentional Database (EDB)* handelt es sich um Relationen, deren Tupel physisch in einer Datenbank als Tabellen gespeichert sind. Bei

der *Intensional Database (IDB)* hingegen findet keine physische Speicherung der Tupel statt. Stattdessen werden die Tupel von anderen Relationen abgeleitet. Im Beispiel 3.1 entspricht das Prädikat $fahrt(x, y)$ einer EDB, die mit einer Tabelle repräsentiert werden kann, und das Prädikat $erreichbar(x, y)$ einer IDB, die mit einer Sicht ($view$) dargestellt werden kann. Die Beantwortung von rekursiven Anfragen wie $erreichbar(x, y)$ in relationalen Datenbanken wurde erst in SQL-99 eingeführt. Sie befolgen die *Least Fixed Point*-Semantik und haben die folgende Syntax:

$$\begin{array}{l} \textbf{WITH} \quad \textbf{RECURSIVE} \ R \ \textbf{AS} \\ \qquad < \textit{DEFINITION} \quad \textit{VON} \quad R > \\ < \textit{ANFRAGE} \quad \textit{ÜBER} \ R > \end{array}$$

Beispielsweise wird die entsprechende SQL-Version des Datalog-Programms vom Beispiel 3.1 wie folgt geschrieben:

$$\begin{array}{l} \textbf{WITH} \ \ \textbf{RECURSIVE} \ erreichbar(dep,\ dest) \ \textbf{AS} \\ \quad \textbf{SELECT} \ dep,\ dest \ \textbf{FROM} \ fahrt \\ \quad \textbf{UNION} \\ \quad \textbf{SELECT} \ fahrt.dep,\ erreichbar.dest \\ \quad \textbf{FROM} \ fahrt,\ erreichbar \\ \quad \textbf{WHERE} \ fahrt.dest = erreichbar.dep \\ \textbf{SELECT} \ dep,\ dest \ \textbf{FROM} \ erreichbar \end{array}$$

Hier liefert die letzte Anfrage (**SELECT** $dep,\ dest$ **FROM** $erreichbar$) die transitive Hülle der binären Relation $fahrt$.

3.2.2 Die transitive Hülle für Entity Resolution

In der letzten Phase eines Data-Matching-Prozesses erfolgt die Evaluierung der Match-Ergebnisse. Es ist offensichtlich, dass jedes einzelne Ergebnispaar in eine der vier Kategorien fällt:

True Positive bedeutet, dass ein Entitäten-Paar, das in der Realwelt ein match darstellt, als match erkannt wurde.

False Positive heißt, dass ein Entitäten-Paar als match klassifiziert wurde, obwohl die Entitäten unterschiedliche Realweltobjekte darstellen.

True Negative sind jene Entitäten-Paare, die als nicht-match erkannt wurden und auch tatsächlich kein match darstellen.

False Negative sind die Entitäten-Paare, die fälschlicherweise als nicht-match klassifiziert wurden, obwohl es sich bei ihnen um ein match handelt.

Zwei Maße, *precision* und *recall*, wurden im Bereich von Information Retrieval definiert. Sie werden bei Data Matching für die Bestimmung der Qualität des Match-Verfahrens bezüglich der Genauigkeit und Vollständigkeit verwendet und sind wie folgt definiert:

$$precision = \frac{True\ Positive}{True\ Positive\ +\ False\ Positive} \tag{3.3}$$

$$recall = \frac{True\ Positive}{True\ Positive\ +\ False\ Negative} \tag{3.4}$$

Die Genauigkeit der Ergebnisse wird von allen Phasen des Match-Prozesses beeinflusst, i. e. Preprocessing, Blockbildung, Ähnlichkeitsberechnung und Klassifizierung. Die Vollständigkeit wird vor allem von der Blockbildung beeinflusst. Grund hierfür ist, dass für die Vermeidung eines quadratischen Vergleiches der Datensätze ein *einfaches* Kriterium verwendet wird, um *"ähnliche"* Datensätze in Blöcken zu clustern, innerhalb derer die Entitäten paarweise miteinander verglichen werden. Die Verwendung eines einfachen Kriteriums für das Blocking kann jedoch dazu führen, dass zwei Entitäten, die ein richtiges match darstellen, zwei verschiedenen Blöcken zugewiesen werden, sodass sie gar nicht miteinander verglichen und als match entdeckt werden. Die Formel (3.4) verdeutlicht den Zusammenhang zwischen den nicht entdeckten matches und dem Wert des Maßes *recall*. Eine hohe Rate an *False Negative* hat einen negativen Einfluss auf das Maß *recall*, das wiederum eine negative Wirkung auf die gesamte Qualität des Match-Verfahrens hat. Außerdem kann das Match-Ergebnis einige Widersprüche enthalten [Chr12]. So ist es wahrscheinlich, dass das Match-Ergebnis M die zwei matches (a, b) und (b, c) enthält, jedoch nicht das match (a, c). Es ist insofern ein Widerspruch, als die Ähnlichkeit eine transitive Eigenschaft[3] ist. Um diese Widersprüche zu beheben und somit das Maß *recall* zu verbessern, wird die transitive Hülle nach der Klassifizierungsphase berechnet. Dabei wird das finale Match-Ergebnis M um die Korrespondenz (a, c) erweitert, sofern $(a, b) \in M$ und $(b, c) \in M$ aber $(a, c) \notin M$. Im nächsten Abschnitt werden verschiedene Algorithmen dargestellt, die die transitive Hülle berechnen.

[3] nur unter bestimmten Voraussetzungen, die in der Evaluierung betrachtet werden

3.3 Sequentielle Algorithmen

Sequentielle Algorithmen zur Berechnung der transitiven Hülle eines Graphen werden in *direkte* und *iterative Algorithmen* [CCH93] unterteilt.

3.3.1 Direkte Algorithmen

Direkte Algorithmen verwenden die Darstellung des Graphen als Matrix und arbeiten *depth-first*. Die Anzahl der Iterationen wird durch die Größe der Matrix beeinflusst und kann schon vor der Berechnung ermittelt werden.

3.3.1.1 Warshall's Algorithmus

Der Ausgangspunkt für Warshall's Algorithmus [War62] ist die Adjazenz-Matrix $M(n \times n)$ des Graphen, wobei n die Anzahl seiner Knoten ist. Die Matrix M ist vom Typ boolean und hat als Eintrag $M_{i,j} = 1$, wenn es eine Kante zwischen den Knoten i und j gibt, sonst $M_{i,j} = 0$.

Die Abb. 3.1 zeigt die Grundidee der Arbeitsweise von Warshall's Algorithmus. In der Iteration k wird eine neue Matrix M^k aus der alten M^{k-1} generiert. Die Generierung erfolgt nach zwei Regeln:

1. Wenn der Eintrag $M_{ij}^{k-1} = 1$, dann bleibt er in M_{ij}^k unverändert, i. e. in jeder Iteration werden keine Kanten gelöscht, sondern nur hinzugefügt.
2. Wenn der Eintrag $M_{ij}^{k-1} = 0$, aber die zwei Einträge $R_{ik}^{k-1} = 1$ und $R_{kj}^{k-1} = 1$, dann wird $M_{ij}^k = 1$ geändert. Diese Änderung bedeutet, dass es einen *Weg* von i zu j via höchstens k Knoten gibt.

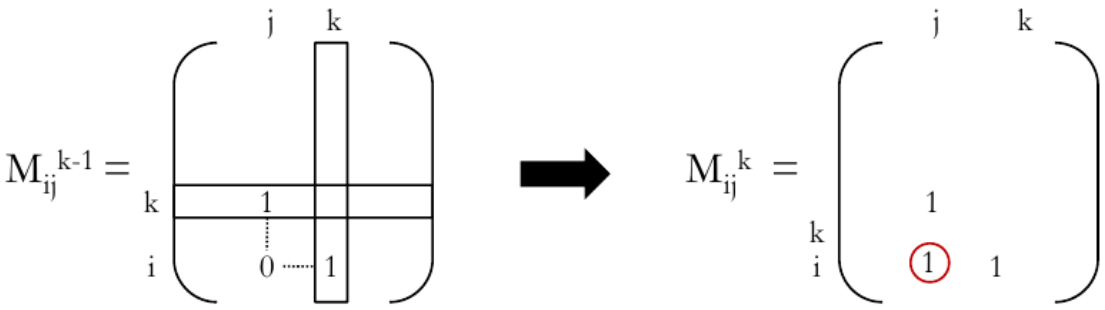

Abbildung 3.1: Arbeitsweise von Warshall's Algorithmus

Algorithm 3.1 Warshall

Input: $M(n \times n)$ // Adjazenz-Matrix des Graphen
Output: $M^*(n \times n)$ // Transitive Hülle des Graphen
1: $R^0 = M$;
2: **for** $k = 1$ to n **do**
3: **for** $i = 1$ to n **do**
4: **for** $j = 1$ to n **do**
5: $R_{ij}^k = R_{ij}^{k-1}$ OR $(R_{ik}^{k-1}$ AND $R_{kj}^{k-1})$;

Warshall's Algorithmus durchläuft die Matrix dreimal, um die transitive Hülle zu berechnen und hat die Laufzeit-Komplexität $\mathcal{O}(n^3)$. Das Problem bei diesem Algorithmus ist die Notwendigkeit der Speicherung der gesamten Relation im Hauptspeicher eines Rechners, was für große Relationen nicht machbar ist. Aus diesem Grund werden solche Algorithmen in dieser Arbeit nicht verfolgt und wurden hier nur der Vollständigkeit halber erwähnt.

3.3.2 Iterative Algorithmen

Iterative Algorithmen umfassen Methoden, die die transitive Hülle mittels algebraischer Operationen berechnen. Sie verwenden dafür die tabellarische Darstellung der Relation und arbeiten *breadth-first*. Bei solchen Algorithmen wird die Anzahl der benötigten Iterationen durch die Tiefe des Graphen bestimmt und ist vor der kompletten Berechnung der transitiven Hülle nicht bekannt.

3.3.2.1 Naiver Algorithmus

Der naive Algorithmus [Ban85] ist ein schlichtes Implementieren der algebraischen Definition der transitiven Hülle. Dabei werden drei algebraische Operationen benutzt, nämlich ein *Join*, eine *Union* und eine *Differenz*. Die Grundidee wird anhand des Algorithmus 3.2 erläutert. Zwei Relationen, P und T, die mit der Ausgangsrelation R initialisiert sind, werden benötigt. T beinhaltet das Ergebnis aller Iterationen und P enthält nur das Ergebnis der letzten Iteration. In der Iteration i wird eine Join-Operation zwischen der *power* Relation P^{i-1}, die in der letzten Iteration berechnet wurde, und der Ausgangsrelation R durchgeführt (Zeile 7). Das Ergebnis dieser Join-Operation wird zur Relation T hinzugefügt (Zeile 8). Anschließend wird eine Differenz zur Überprüfung des Vorhandenseins von neuen Tupeln durchgeführt (Zeile 9). Falls neue Tupel entdeckt wurden, wird der Prozess wie beschrieben fortgesetzt, sonst terminiert

Algorithm 3.2 Naive

Input: $R(x, y)$ // Ausgangsrelation
Output: $T(x, y)$ // Transitive Hülle von R
1: $P^0 = R$; // Power Relation für das Ergebnis einer Iteration
2: $T = R$;
3: $i = 0$;
4: **repeat**
5: $i = i + 1$;
6: $T_{old} = T$;
7: $P^i = \pi_{(P^{i-1}.x, R.y)}(P^{i-1} \bowtie_{(P^{i-1}.y = R.x)} R)$;
8: $T = T \cup P^i$;
9: **until** $(T \setminus T_{old} = \emptyset)$
10: **return** T;

der Algorithmus mit der Ausgabe von T als transitive Hülle der Relation R. Bei der Darstellung von R mit einem Graphen entspricht das Ergebnis einer Iteration i allen Pfaden mit einer Länge $\leqslant i$.

Dieser Algorithmus hat drei Nachteile: 1) In jeder Iteration werden alle Ergebnisse der vorherigen Iterationen in der Join-Operation verwendet. 2) Bei zyklischen Graphen werden Duplikate erzeugt aber nicht entfernt, was zu redundanter Berechnung führt. 3) Dieser Algorithmus ist *linear* und konvergiert nach d Iterationen, wobei d die Tiefe des Graphen ist, i. e. die größte absolute Entfernung zwischen zwei Knoten.

3.3.2.2 Semi-naiver Algorithmus

Der Semi-naive Algorithmus [BMSU86] ist eine Verbesserung des naiven. Dabei wird versucht, die zwei ersten Nachteile des naiven Algorithmus zu beheben, indem in der aktuellen Iteration nur das Ergebnis der *letzten* Iteration benutzt wird. In Zeile 7 des Algorithmus 3.3 wird eine Differenz zwischen den Relationen P^i, die das Ergebnis der Join-Operation beinhaltet, und T durchgeführt. Diese wird in D^i gespeichert. Demzufolge beinhaltet D^i bei der Ausführung der nächsten Iteration nur die neu entdeckten Tupel. Die Differenz dient auch der Überprüfung der Terminierung des Algorithmus (Zeile 9).

Algorithm 3.3 Semi-naive

Input: $R(x, y)$ // Ausgangsrelation
Output: $T(x, y)$ // Transitive Hülle von R
1: $D = R$; // Differenz-Relation
2: $T = R$;
3: $i = 0$;
4: **repeat**
5: $i = i + 1$;
6: $P^i = \pi_{(D^{i-1}.x, R.y)}(D^{i-1} \bowtie_{(D^{i-1}.y = R.x)} R)$;
7: $D^i = P^i \setminus T$;
8: $T = T \cup D^i$;
9: **until** $(D^i = \emptyset)$
10: **return** T;

Obwohl der Semi-naive Algorithmus eine große Verbesserung hinsichtlich der Reduzierung der Datenmenge und redundanter Berechnung darstellt, hat er immer noch den Nachteil, dass er nach d Iterationen konvergiert.

3.3.2.3 Squaring Algorithmus

Das Ziel von Squaring Algorithmus [AHB86] ist die Reduzierung der Anzahl von Iterationen für ein schnelleres Konvergieren. Der Squaring Algorithmus hat fast denselben Ausführungsplan wie die vorherigen Algorithmen, mit dem Unterschied, dass er, statt einen Join zwischen der Ausgangsrelation und dem Ergebnis der letzten Iteration auszuführen, einen *self-Join* des Ergebnisses der letzten Iteration ausführt.

Algorithm 3.4 Squaring

Input: $R(x, y)$ // Ausgangsrelation
Output: $T(x, y)$ // Transitive Hülle von R
1: $T = R$;
2: **repeat**
3: $T_{old} = T$;
4: $T = \pi_{(T.x, T.y)}(T \bowtie_{(T.y = T.x)} T)$;
5: $T = T \cup R$;
6: **until** $(T \setminus T_{old} = \emptyset)$
7: **return** T;

Die Anwendung der Squaring-Methode für die Berechnung der transitiven Hülle eines Graphen führt zur Entdeckung von Pfaden, deren Längen in der Iteration i bis maximal 2^i betragen können. Beispielsweise werden in der dritten Iteration Pfade mit Längen bis zu $2^3 = 8$ entdeckt. Das bedeutet, dass dieser Algorithmus nicht linear, wie die zwei ersten, sondern *logarithmisch* konvergiert, i. e. er terminiert nach $(\log d) + 1$ Iterationen. Der Squaring Algorithmus hat trotzdem den Nachteil, dass eine sehr große Anzahl von Duplikaten erzeugt wird und dass dies zu einer erheblich redundanten Berechnung bei großen Datenmengen führt.

3.3.2.4 Smart Algorithmus

Der Smart Algorithmus wurde in [Ioa86] vorgeschlagen. Er stellt hinsichtlich der Anzahl der erzeugten Duplikate eine Verbesserung des Squaring Algorithmus dar. Die Grundidee von Smart ist die Erstellung eines Pfades aus einem *Präfix*- und einem *Suffix*-Pfad, unter Voraussetzung bestimmter Pfadlängen. Bedingung dabei ist, dass das *Präfix* nie kleiner als das *Suffix* ist. Der Algorithmus 3.5 verwendet dafür in einer Iteration i zwei Join-Operationen. Die erste entdeckt Pfade mit Längen zwischen 2^{i-1} und $2^i - 1$ (Zeile 6), so werden z. B. in Iteration 3 Pfade mit den Längen $4, 5, 6, 7$ entdeckt. Die zweite Join-Operation generiert Pfade, deren Länge genau 2^i (2, 4, 8, ...) beträgt (Zeile 8). Diese Pfade werden immer aus einem Präfix- und einem Suffix-Pfad der gleichen Länge erzeugt. Zusätzlich zu den Joins werden eine Union und eine Differenz für die Eliminierung von Duplikaten durchgeführt. Der Algorithmus terminiert nach $(\log d) + 1$ Iterationen.

Algorithm 3.5 Smart

Input: $R(x, y)$ // Ausgangsrelation
Output: $P(x, y)$ // Transitive Hülle von R
1: $Q_0 = R$;
2: $P_0(x, x) = \{(x, x)|$x is a graph node$\}$;
3: $i = 0$;
4: **repeat**
5: $i++$;
6: $P_i = Q_{i-1} \bowtie P_{i-1}$;
7: $P_i = P_i \cup P_{i-1}$;
8: $Q_i = Q_{i-1} \bowtie Q_{i-1}$;
9: $Q_i = Q_i \setminus P_i$;
10: **until** $Q_i = \emptyset$
11: **return** P_i;

3.4 Parallele Algorithmen

Die Parallelisierung von der Berechnung der transitiven Hülle ist eine Art *Intra-Query-Parallelisation* [CCH93], i. e. eine einzige Anfrage wird auf mehreren Rechner ausgeführt. Die Voraussetzung dafür ist eine Partitionierung der originalen Relation auf mehrere Rechner. Außerdem müssen die unterschiedlichen Operationen Join, Union und Differenz zwischen diesen Rechnern synchronisiert werden. Im Folgenden werden zwei Methoden der Parallelisierung vorgestellt. Die erste Methode ist Hash-basiert und die zweite ist ein Vorschlag für die Verwendung von MapReduce bei der Berechnung der transitiven Hülle.

3.4.1 Hash-basierte Parallelisierung

3.4.1.1 Transitive Closure with Parallel Operations

Der Algorithmus *Transitive Closure with Parallel Operations* (TCPO) [VK88] ist eine Art parallele Implementierung des naiven Algorithmus [Ban85], bei der die Join- und Union-Operationen auf mehreren Rechnern ausgeführt werden. Eine Hash-Funktion h wird verwendet, um Relationen auf die verfügbaren Knoten des Clusters zu partitionieren. Für eine binäre Relation $R(x, y)$ wird diese Funktion auf das zweite Attribut y angewandt. Dieselbe Hash-Funktion wird ebenfalls auf eine Kopie $D(x, y)$ von R angewandt, diesmal jedoch auf das erste Attribut x von D. Das Resultat sind die Partitionen R_1, R_2,...,R_i von R und D_1^1, D_2^1,...,D_i^1 von D, welche den entsprechenden Rechnern 1, 2,...,i zugewiesen werden. In der ersten Iteration wird in jedem Rechner i ein Join $D_i^2 = \pi_{(R_i.x, D_i.y)}(R \bowtie_{(R.y=D_i.x)} D_i^1)$ durchgeführt. Das Ergebnis D_i^2 wird für die nächste Iteration wieder partitioniert, indem die Hash-Funktion h auf das erste Attribut x von D_i^2 angewandt wird. Grund hierfür ist, dass die Projektion die Werte des Join-Attributs x der Relation D_i^2 verändert hat. In der folgenden Iteration wird $D_i^3 = \pi_{(R_i.x, D_i^2.y)}(R \bowtie_{(R.y=D_i^2.x)} D_i^2)$ berechnet, wobei D_i^2 die dem Rechner i zugewiesenen Tupel nach der Umverteilung darstellt. Die bereits gespeicherten Partitionen R_i werden in jeder Iteration ohne Umverteilung wiederverwendet. In jeder Iteration j fügt der Rechner i die neu generierten Paare D_i^{j+1} zu TC_i, die mit den Partitionen R_i initialisiert wurde. Der Prozess wird so lange fortgesetzt, bis keine neuen Paare mehr generiert werden, i. e. nach der j-ten Iteration ist $\bigcup_{i=1}^{n} D_i^{j+1} = \emptyset$.

Algorithm 3.6 TCPO

Input: $R(x, y)$ // Ausgangsrelation
Output: $T(x, y)$ // Transitive Hülle von R
1: $D^1 = R$;
2: partition $(R, h(y))$; // Partitionierung von R in R_1, R_2,...,R_i
3: partition $(D^1, h(x))$; // Partitionierung von D in D_1^1, D_2^1,...,D_i^1
4: $T_i = R_i$; // Initialisierung von T
5: $j = 1$;
6: **repeat**
7: **for all** $i \in \{1, \cdots, n\}$ **do**
8: $D_i^{j+1} = \pi_{(R_i.x, D_i^j.y)}(R_i \bowtie_{(R_i.y=D_i^j.x)} D_i^j)$;
9: $T_i = T_i \cup D_i^{j+1}$;
10: partition $(D_i^{j+1}, h(x))$;
11: $j = j + 1$;
12: **end**
13: **until** $(\bigcup_{i=1}^n D_i^{j+1} = \emptyset)$
14: **return** $\bigcup_{i=1}^n T_i$;

Der obere Algorithmus ist eine vereinfachte Version und setzt für sein Konvergieren einen azyklischen Graphen voraus. Wenn der Graph jedoch Zyklen enthält, muss nach jeder j-ten Iteration $\bigcup_{i=1}^n D_i^{j+1} \setminus \bigcup_{i=1}^n D_i^j$ berechnet werden um sicherzustellen, dass der Algorithmus TCPO nach d terminiert.

3.4.1.2 Double Hash Transitive Closure

Einer der größten Nachteile von TCPO ist, dass das Teilergebnis einer Iteration, das auf einem Rechner liegt, mittels der Hash-Funktion h auf die entsprechenden Rechner des Clusters umverteilt werden muss. Als Lösung für das Problem wurde *Double Hash Transitive Closure* DHTC in [CdM89] präsentiert. Die Grundidee ist die Verwendung von *double-hashing*, um die Ausgangsrelation bezüglich mehrerer Attribute zu verteilen und zu clustern. Der Einfachheit halber wird eine Hash-Funktion h benutzt, die jedes Tupel einer Relation auf einer natürlichen Zahl n abbildet. Für ein Cluster bestehend aus k Rechnern liefert h Hashwerte zwischen 0 und $k-1$, die den Ids dieser Rechner entsprechen. h wird zunächst auf das zweite Attribut y der binären Relation $R(x, y)$ angewandt (Zeile 2). Die daraus resultierenden Partitionen R_0, R_1,...,R_{k-1} werden dann auf die entsprechenden Rechner 1, 2,$\cdots$,$k-1$ verteilt. Danach wird jede von diesen Partitionen in Subrelationen *geclustert*, indem dieselbe Hash-Funktion h auf das erste Attribut x angewandt wird (Zeile 7). Das Resultat der doppelten Verwendung von Hashing auf die Relation R sind die Subrelationen R_{ij} mit

$0 \leqslant i < k-1$ und $0 \leqslant j < k-1$, wobei der Index i die geclusterten Subrelationen einer einzigen Partition auf einem Rechner bezeichnet und der Index j alle auf den Rechnern verteilten Partitionen darstellt. Außerdem wird für die erste Iteration eine Kopie von R, die Relation $D(x, y)$, in D_0^1, D_1^1,...,D_{k-1}^1 partitioniert und auf die entsprechenden Rechner verteilt. Das geschieht durch die Anwendung von h auf das erste Attribut x von D (Zeile 3). Jeder Rechner j führt Join-Operationen zwischen den Teilmengen der Partition R_j und der Partition D_j aus, i. e. $D_{ij}^2 = \sum_{i=0}^{k} \pi_{(R_{ij}.x, D_j.y)}(R_{ij} \bowtie_{(R_{ij}.y = D_j.x)} D_j^1)$ (Zeile 12). Das Teilergebnis D_{ij}^2 im Rechner j ist in Subrelationen nach dem Attribut x geclustert (i ist der Index jeder Subrelation). Für die nächste Iteration wird kein Hashing mehr benötigt. Stattdessen sendet der Rechner j jede Subrelation D_{pj}^2 zum Rechner p (Zeile 13) und empfängt von jedem Rechner p die Subrelation D_{jp}^2 (Zeile 15).

Algorithm 3.7 DHTC

Input: $R(x, y)$ // Ausgangsrelation
Output: $T(x, y)$ // Transitive Hülle von R

1: $D^1 = R$;
2: partition $(R, h(y))$; // Partitionierung von R in R_0, R_1,...,R_{k-1}
3: partition $(D^1, h(x))$; // Partitionierung von D in D_0^1, D_2^1,...,D_{k-1}^1
4: $T_i = R_i$; // Initialisierung von T
5: $n = 1$;
6: **for all** $i \in \{0, \cdots, k-1\}$ **do**
7: cluster $(R_i, h(x))$; // Jede Partition i wird nach x geclustert
8: **end**
9: **repeat**
10: **for all** $j \in \{0, \cdots, k-1\}$ **do**
11: **for all** $i \in \{0, \cdots, k-1\}$ **do**
12: $D_{ij}^{n+1} = \pi_{(R_{ij}.x, D_j^n.y)}(R_{ij} \bowtie_{(R_{ij}.y = D_j^n.x)} D_j^n)$;
13: send D_{ij}^{n+1} to node i; $0 \leqslant i < k-1 \ \wedge \ i \neq j$
14: **end**
15: recieve D_{jp}^{n+1} from all nodes p; $0 \leqslant p < k-1 \ \wedge \ p \neq j$
16: $D_j^{n+1} = \bigcup_{r=0}^{k-1} D_{jp}^{n+1}$;
17: $T_j = T_j \bigcup D_j^{n+1}$;
18: $n = n + 1$;
19: **end**
20: **until** $(\bigcup_{j=0}^{k-1} D_j^{n+1} = \emptyset)$
21: **return** $\bigcup_{j=1}^{k-1} T_j$;

Derselbe Prozess (ohne Hashing) wird fortgesetzt, bis in allen Rechnern keine neuen Tupel generiert werden. DHTC konvergiert genauso wie TCPO nach d Iterationen.

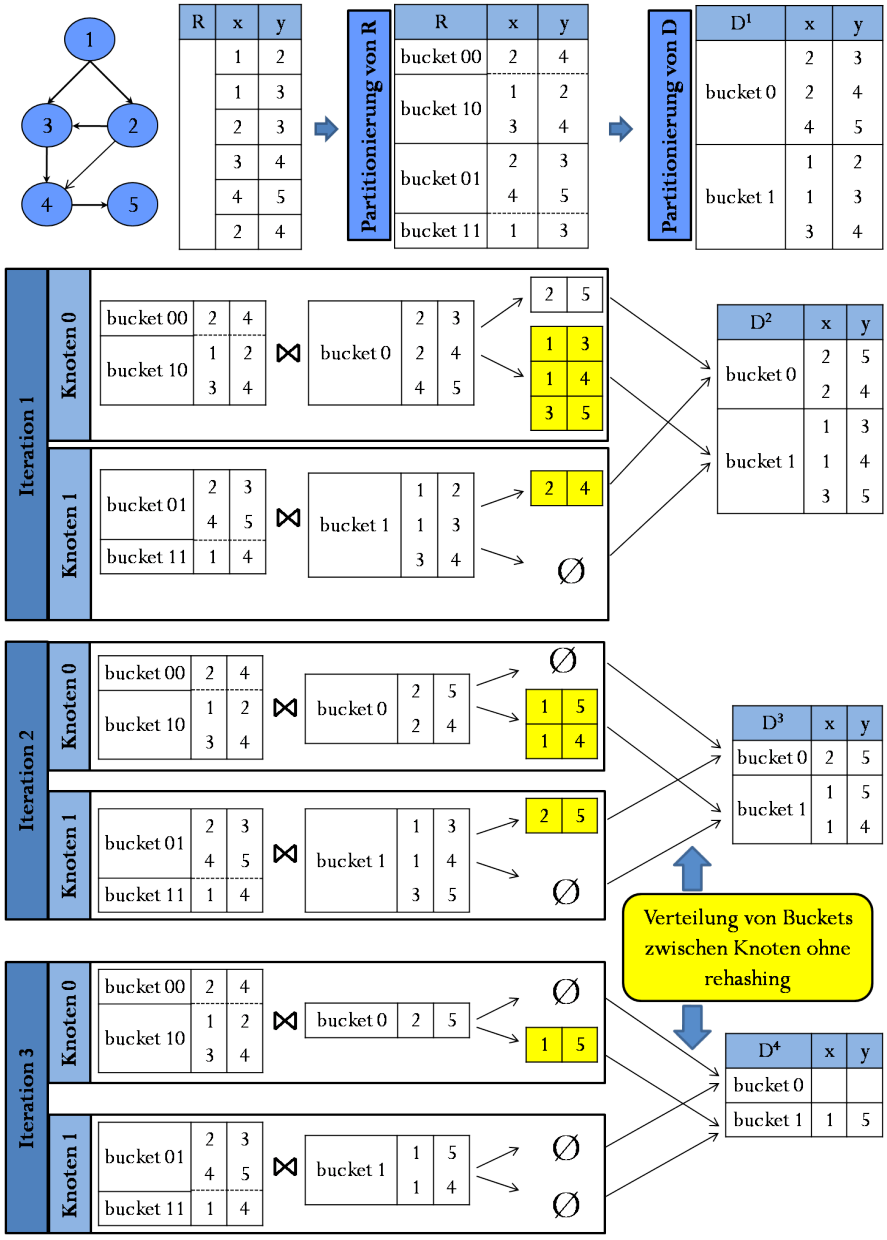

Abbildung 3.2: Verteilte Ausführung von drei Iterationen des DHTC-Algorithmus auf zwei Knoten

Die Abbildung 3.2 verdeutlicht die Arbeitsweise von DHTC mit einer kleinen Relation. Die Ausgangsrelation R wird nach dem Attribut y partitioniert, dann

wird jede Partition nach dem Attribut x geclustert. Das Resultat sind die zwei Subrelationen $bucket00$ und $bucket10$, die zusammen die Partition R_0 darstellen, und die zwei anderen Subrelationen $bucket01$ und $bucket11$, welche die andere Partition R_1 darstellen. Die Relation D (eine Kopie von R) wird nur nach dem Attribut x partitioniert und ergibt die zwei Partitionen $bucket0$ und $bucket1$, die jeweils D_0 und D_1 repräsentieren. Danach werden die Join-Operationen in verschiedenen Knoten parallel ausgeführt. Der Knoten 0 führt die Join-Operation zwischen R_0 und D_0^1 ($bucket0$) aus. Konkret heißt das: zwischen $bucket00$ und D_0^1 und zwischen $bucket10$ und D_0^1. Das Ergebnis dieser Join-Operationen sind zwei nach x geclusterte Subrelationen D_{00}^2 mit dem Tupel $\{(2,5)\}$ und D_{10}^2 mit den Tupeln $\{(1,3),(1,4),(3,5)\}$. Anschließend sendet der Knoten 0 die Subrelation D_{10}^2 zum Knoten 1, weil der erste Index $i = 1$ mit der Id des Knoten 1 übereinstimmt, und empfängt von ihm die Subrelation D_{01}^2 mit dem Tupel $\{(2,4)\}$ aus demselben Grund. Die nächsten Iterationen erfolgen analog zur ersten ohne Anwendung der Hash-Funktion.

3.4.2 MapReduce-basierte Parallelisierung

Die parallele Berechnung der transitiven Hülle mit MapReduce wurde in [ABC+11] umrissen, ohne Implementierungsdetails oder Auswertungen zu präsentieren. Für eine nicht-lineare (logarithmische) Berechnung der transitiven Hülle mittels MapReduce benötigt die Implementierung zwei Hash-Funktionen h und g und zwei Arten von Tasks, *Join-Task* und *Dup-elim-Task*, deren Aufgaben die Folgenden sind:

- Hash-Funktion h: Diese Funktion wird auf jedes gelesene Paar (x, y) angewandt und erzeugt zwei Hashwerte, $h(x)$ und $h(y)$, sodass das Paar (x, y) zwei Partitionen zugewiesen wird.
- Hash-Funktion g: Diese Funktion wird ebenso auf jedes Paar (x, y) angewandt, jedoch nur auf das erste Attribut x. Dann wird das Paar zwecks Überprüfung auf ein vorhandenes Duplikat zu der entsprechenden Partition gesendet.
- Join-Tasks: Ein Join-Task i empfängt Paare (x, y), wenn $h(x) = i$ oder $h(y) = i$ und speichert sie lokal. Er generiert danach das Paar (x, z), falls die zwei Paare (x, y) und (y, z) in der Partition existieren.
- Dup-elim-Tasks: Ein Dup-elim-Task j empfängt jedes Paar (x, y), dessen Hashwert $h(x) = j$, und überprüft, ob dasselbe Paar schon vorhanden ist. In diesem Falle wird das neue Paar ignoriert, andernfalls speichert er es lokal.

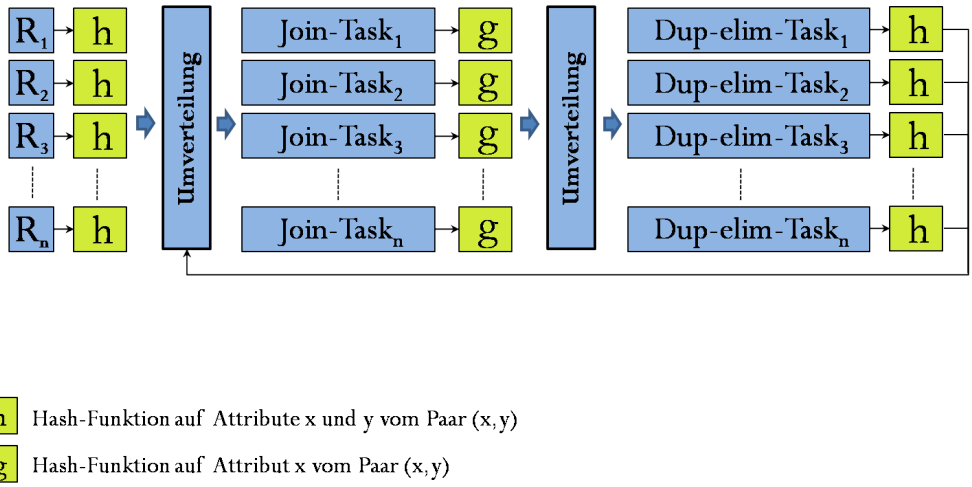

Abbildung 3.3: Der Ausführungsplan einer nicht-linearen verteilten Methode zur Berechnung der transitiven Hülle mit MapReduce

Die Ausführung erfolgt in einer Schleife und wird in Abbildung 3.3 präsentiert. Die Hash-Funktion h liest erstmals die Ausgangsrelation und verteilt die Daten bezüglich beider Attribute auf die Join-Tasks. Diese generieren neue Paare, die das Input der Hash-Funktion g darstellen. Die Anwendung von g auf das erste Attribut des Paares verteilt diese Daten auf die Dup-elim-Tasks. Diese überprüfen, ob Duplikate erzeugt wurden. Dann geben sie die duplikatfreien Paare für die Hash-Funktion h aus.
Da diese Art der Berechnung trotzdem eine große Anzahl von Duplikaten generiert und eine redundante Berechnung verursacht, wurde zum Beheben des Problems eine Variante von Smart-Algorithmus vorgeschlagen. Smart, wie im Abschnitt 3.3.2.4 beschrieben, führt zwei Join-Operationen aus. In einer Iteration i entdeckt der erste Join Pfade, deren Länge zwischen 0 und $2^i - 1$ liegen, und der zweite Join entdeckt Pfade mit der Länge 2^i. Überdies werden eine Union und eine Differenz ausgeführt. Die oben beschriebenen Tasks können bei der Umsetzung des Smart-Algorithmus in MapReduce verwendet werden. Die *Join-Tasks* werden dabei für die Ausführung des ersten Joins benutzt, und die *Dup-elim-Tasks* realisieren die Differenz. Für den zweiten Join wird eine Gruppe von Tasks, die die *Join-Tasks* ähneln, verwendet.

Der vorgeschlagene Ansatz hat jedoch zwei Probleme. Wie die Abbildung 3.4 zeigt, erfordert die zweite Join-Operation ($Q_i = Q_{i-1} \bowtie Q_{i-1}$) eine Umverteilung der Daten oder alternativ eine Replikation einer Teilmenge der Eingabedaten in der map-Phase, weil Paare mit demselben Join-Attribut aufgrund der ersten Partitionierung in verschiedenen Partitionen liegen können. In Abbildung 3.4 liegen beispielsweise die zwei Paare $(1, 3)$ und $(3, 4)$ von

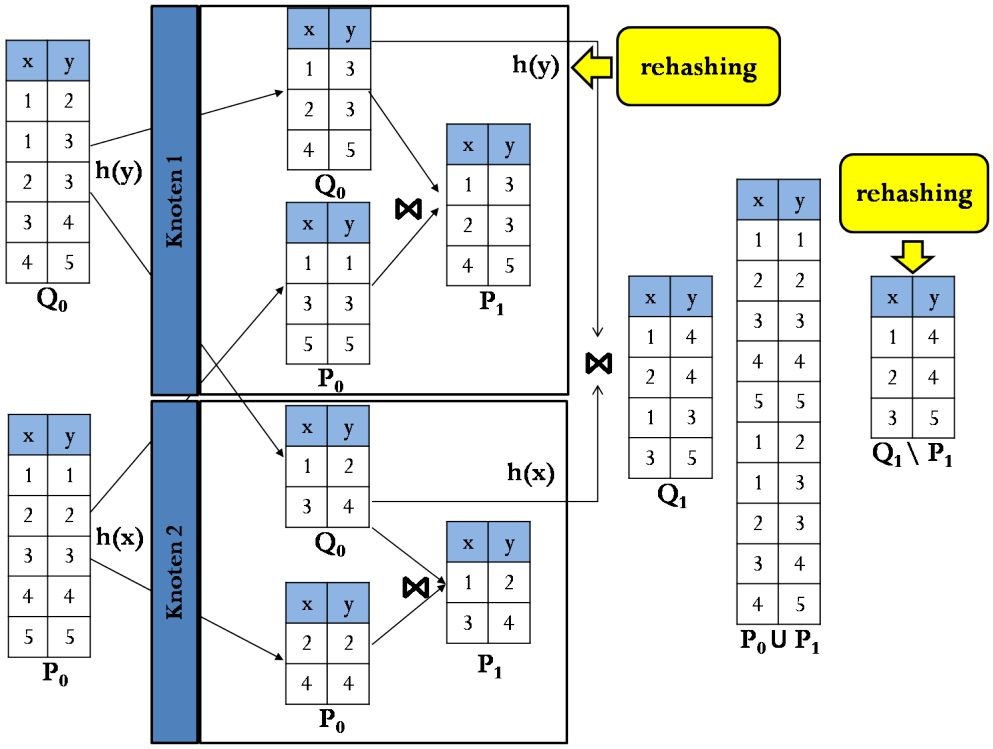

Abbildung 3.4: Die erste Iteration einer verteilten Ausführung von Smart auf zwei Knoten und das Problem der Umverteilung der Daten vor dem zweiten Join und der Differenz

der Relation Q_0, die verbunden werden sollen, in verschiedenen Partitionen. Gleiches gilt für die Berechnung der Differenz. Da eine Umverteilung der Daten einen eigenen MapReduce-Job bedingt, werden für jede Iteration des Algorithmus drei MapReduce-Jobs gestartet. Der Algorithmus terminiert damit nicht nach $(\log d)+1$ wie Smart, sondern nach $3(\log d)+1$ MapReduce-Jobs. Mit einem anderen Framework wie *PACT* [BEH+10], wäre es möglich, diese zwei Tasks (Berechnung der transitiven Hülle und Entfernung der Duplikate) in nur einem Job pro Iteration durchzuführen. *PACT* hat neben map- und reduce-Funktionen drei weitere Funktionen, die ebenso parallel ausgeführt werden können. Der Unterschied zwischen *MapReduce* und *PACT* liegt darin, dass Erstgenanntes aus map- und reduce-Funktionen besteht, die nur in dieser Reihenfolge ausgeführt werden können[4]. Die Funktionen von *PACT* hingegen können in beliebiger Reihenfolge ausgeführt werden, solange sie ein *Directed Acyclic Graph (DAG)* darstellen. Eine von diesen Funktionen ist *CoGroup*, die *(key,value)*-Paare aus

[4] Hadoop bietet auch mapChain und reduceChain, die mehrere map-Funktionen vor und nach der reduce-Phase ausführen lassen. Da die Daten dabei nicht umverteilt werden, hilft das bei der Berechnung der transitiven Hülle nicht.

allen Partitionen im Cluster nach Key gruppiert. Ihre Ausführung nach der reduce-Phase bei der Berechnung der transitiven Hülle hätte die Entdeckung von Duplikaten in derselben Iteration ermöglicht. Da das Tool Dedoop, in das die transitive Hülle integriert wird, MapReduce verwendet, ist die Benutzung eines anderen Frameworks nur schwer realisierbar.

Die dargestellten Algorithmen zeigen die Schwierigkeiten bei der Berechnung der transitiven Hülle. Im Falle von azyklischen Graphen erzeugen die linearen Algorithmen keine Duplikate, aber sie konvergieren bei einem Graphen der Tiefe d nach d Iterationen. Die bevorzugten logarithmischen Algorithmen konvergieren zwar schneller, erzeugen jedoch eine Vielzahl von Duplikaten, deren Entfernung mehr Aufwand verursachen kann. Im nächsten Kapitel wird ein Algorithmus präsentiert, der die transitive Hülle eines azyklischen Graphen ohne Erzeugung von Duplikaten in logarithmischer Zeit berechnet. Für zyklische Graphen werden andere Varianten vom diesem Algorithmus dargestellt.

4 Transitive Hülle mit MapReduce

Das MapReduce-Framework eignet sich aufgrund seiner Art der Parallelisierung hervorragend für die Durchführung von datenintensiven Aufgaben. Seine Verwendung für die Berechnung der transitiven Hülle stellt sich allerdings aufgrund der Umverteilung der Daten und des Entfernens der Duplikate als echte Herausforderung dar. In diesem Kapitel wird zum einen die Grundidee der Berechnung der transitiven Hülle mit MapReduce vorgestellt, zum anderen werden die daraus resultierenden Probleme analysiert. Anschließend werden verschiedene MapReduce-Algorithmen zur Lösung dieser Probleme für unterschiedliche Graphen präsentiert.

4.1 Grundidee, Probleme und deren Lösungen

4.1.1 Grundidee

Die Grundidee für die Berechnung der transitiven Hülle mit MapReduce ist konzeptionell simpel. Die binäre Ausgangsrelation $R(x, y)$ ist initial in mehreren Rechnern im Cluster als Blöcke partitioniert. In der ersten Iteration werden auf jedem Rechner in der map-Phase für ein Eingabe-Tupel (x, y) zwei Key-Value-Paare, (x, y) und (y, x), ausgegeben. Diese map-Ausgabepaare werden dann über die Rechner des Clusters umverteilt, sodass in der reduce-Phase alle Paare mit demselben Key demselben reduce-Task zugewiesen werden. Dieser gruppiert alle Paare, die den gleichen Schlüssel haben, in der Form $(key, List(values))$. In der reduce-Funktion wird die Menge aller Paare von Werten $\{(a, b) \mid a, b \in List(values) \wedge a<b\}$ ausgegeben. Die Vorgehensweise wird anhand der drei rot markierten Knoten B, C, D des Graphen in Abbildung 4.1 erläutert. In der map-Phase liest der erste map-Task das Eingabepaar (C, D) und generiert dafür die Ausgabepaare (C, D) und (D, C). Für das Paar (B, C) gibt der zweite map-Task die Paare (B, C) und (C, B) aus. In der reduce-Phase werden die Paare (C, B) und (C, D) beim reduce-Task$_2$ in der Form $(C, [B, D])$ gruppiert, weil sie denselben Key C haben. Abschließend wird das Paar (B, D) ausgegeben.

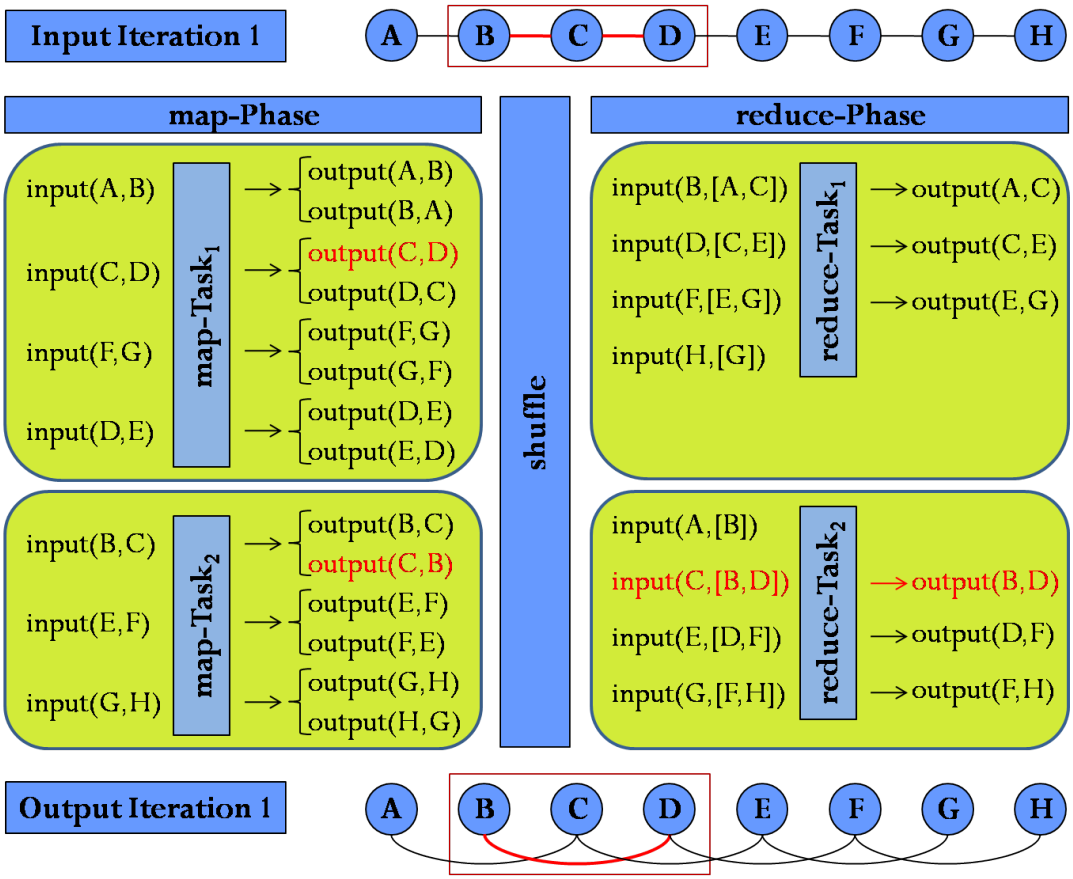

Abbildung 4.1: Die Grundidee der Berechnung der transitiven Hülle mit MapReduce auf zwei Knoten-Cluster

Die strikte Fortsetzung dieses Prozesses erzeugt jedoch eine große Anzahl von Duplikaten. Diese führen zu zwei Problemen. Erstens steigt die Datenmenge in jeder Iteration exponentiell, da diese Duplikate wiederum Duplikate in der nächsten Iteration generieren. Zweitens, wenn die Duplikate nicht als solche erkannt worden sind, kann es geschehen, dass die Berechnung der transitiven Hülle gar nicht terminiert. Eine mögliche Lösung des Problems durch Verwendung der Differenz wie in [ABC+11] ist aus folgendem Grund unzureichend: In Hadoop werden die Daten nach der reduce-Phase in das verteilte Dateisystem geschrieben. Dadurch kann es vorkommen, dass zwei Paare, die ein Duplikat darstellen, auf verschiedenen Rechnern im Cluster gespeichert werden. Um die Daten miteinander zu vergleichen und die Duplikate zu entfernen, muss ein neuer MapReduce-Job gestartet werden, in dessen map-Phase die Daten umverteilt werden. In der sich anschließenden reduce-Phase werden dann die Duplikate entfernt. Diese Konzeption führt zu zwei MapReduce-Jobs, die in jeder Iteration sukzessive ausgeführt werden müssen. Außer der Schwierigkeit

bei der Umsetzung dieser Lösung in MapReduce ist der Overhead beim Starten eines zusätzlichen Jobs pro Iteration inakzeptabel. Aus diesem Grund versucht die vorliegende Arbeit, die Ursachen für die Erzeugung von Duplikaten zu analysieren und zu beheben.

4.1.2 Probleme und deren Lösungen

Für einen ungerichteten azyklischen Graphen wurden folgende drei Gründe für die Erzeugung von Duplikaten bei der Berechnung der transitiven Hülle identifiziert:

Path Decomposition-Problem: Dieses wird anhand der rot eingerahmten vier Knoten C, D, E, F und deren Verbindungen nach der ersten Iteration dargestellt (Abbildung 4.2).

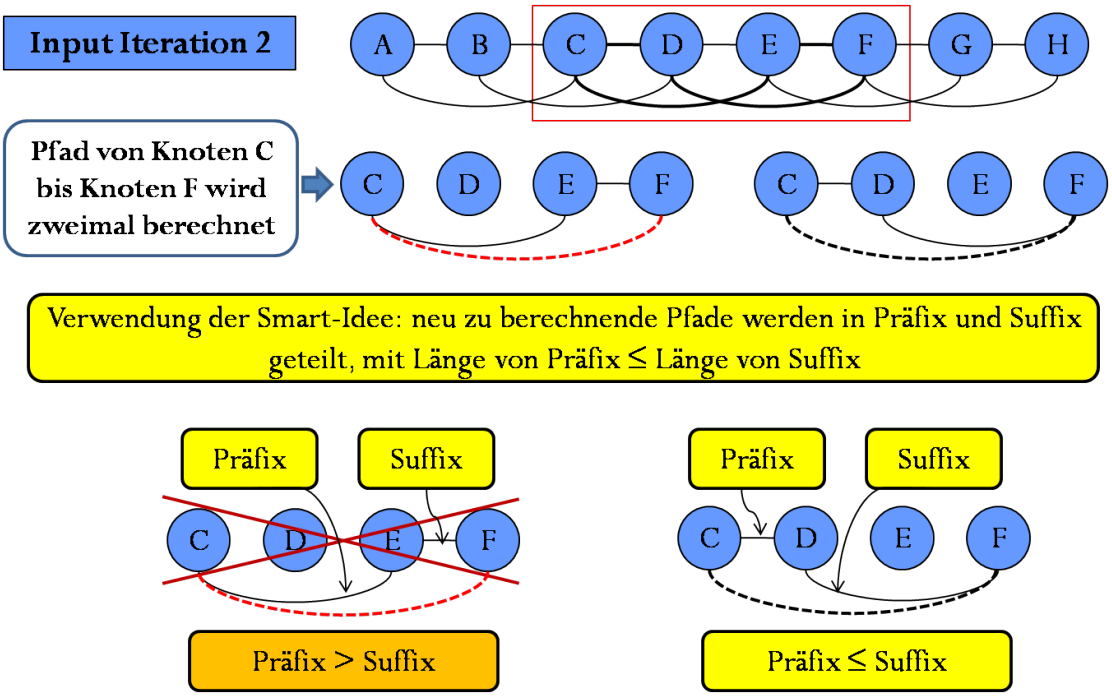

Abbildung 4.2: Path Decomposition-Problem und Lösungsansatz

Die zweite Iteration erhält die Ausgangsrelation R sowie das Ergebnis der ersten Iteration als Eingabe. Mit der Anwendung der Grundidee kann man ausgehend vom Knoten C den Knoten F über zwei Wege erreichen. Zum einen ist F über (C, E) und (E, F) und zum anderen über (C, D) und (D, F) erreichbar. Die Folge ist eine redundante Erzeugung des Paares (C, F). Diesem Problem

kann mit der Idee des Smart-Algorithmus begegnet werden. Der zu konstruierende Pfad von C nach F wird in einen *Präfix*- und einen *Suffix*-Pfad mit bestimmten Längeneigenschaften geteilt. Der Präfix-Pfad darf dabei nicht länger als der Suffix-Pfad sein (Eine umgekehrte Festlegung wäre ebenso denkbar.). Außerdem darf die Differenz der beiden Längen nicht größer als 1 sein. Im Beispiel der Abbildung 4.2 besteht das Präfix aus der Kante $C-D$ und das Suffix aus dem Pfad $D-E-F$, die zusammen den neuen Pfad von C nach F mit der Länge 3 erzeugen.
Es wurde bereits erwähnt, dass ein zu konstruierender Pfad in ein Präfix und ein Suffix geteilt wird, wobei u.a. der Längenunterschied zwischen ihnen ≤ 1 ist. Es gibt jedoch eine Ausnahme: Wenn der Präfix-Pfad aus der Ausgangsrelation (mit Länge 1) und der Suffix-Pfad aus dem Ergebnis der letzten Iteration $i-1$ (mit maximaler Länge 2^{i-1}) stammen, so erzeugen sie in der Iteration i stets einen Pfad mit der Länge $2^{i-1}+1$. In der dritten Iteration der Abbildung 4.3 beispielsweise gibt es keine andere Möglichkeit, einen Pfad zwischen B und G mit der Länge 5 zu erzeugen, außer der Verwendung von $B-C$ mit Länge 1 (aus der Ausgangsrelation) und dem Pfad von C bis G mit Länge 4 (aus der zweiten Iteration).

Short Path-Problem: Bei der iterativen Berechnung der transitiven Hülle mit logarithmischem Verhalten werden in der Iteration i Pfade entdeckt, deren Länge maximal 2^i betragen kann. Dies beinhaltet jedoch kürzere Pfade, die bereits in Iteration $j < i$ entdeckt wurden.

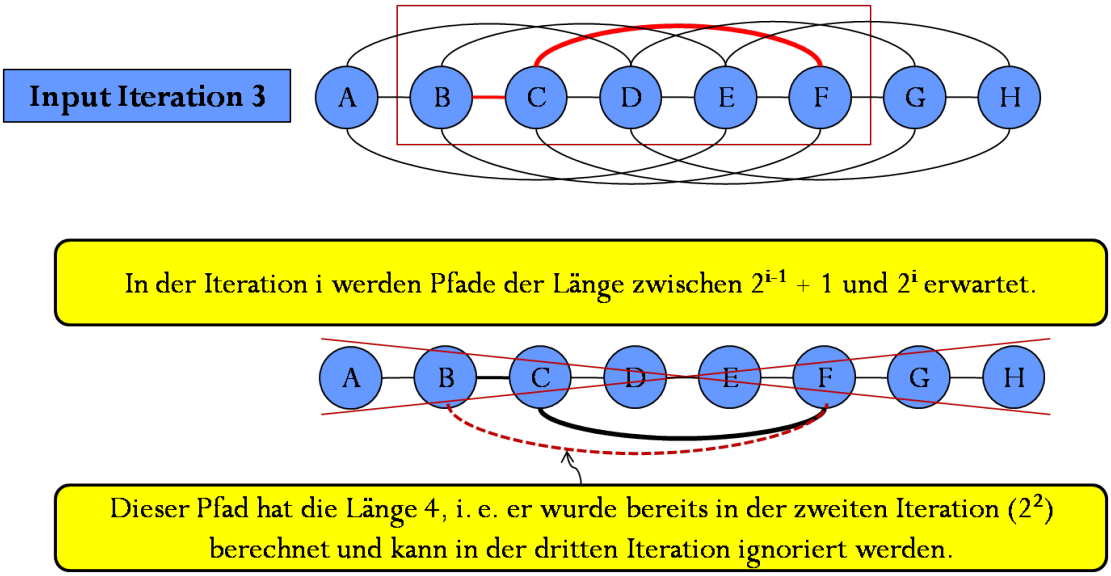

Abbildung 4.3: Short Path-Problem und seine Lösung

Die Abbildung 4.3 zeigt dies anhand der dritten Iteration des laufenden Beispiels. Dabei wird der Pfad mit der Länge 4 vom Knoten B nach F über C konstruiert ($B{-}C{-}D{-}E{-}F$), der bereits in der zweiten Iteration generiert wurde ($4 = 2^2$). Zur Lösung des Problems wird in der Iteration i für jeden berechneten Pfad der Länge l überprüft, ob $2^{i-1}{+}1 \leq l \leq 2^i$. Ist dies nicht der Fall, wird der Pfad verworfen. Demzufolge werden in der dritten Iteration Pfade mit einer Länge zwischen $2^{3-1}{+}1 = 5$ und $2^3 = 8$ konstruiert und der Pfad $B{-}C{-}D{-}E{-}F$ ignoriert.

Direction-Problem: Für ungerichtete Graphen existiert ein weiteres Problem, das in Abbildung 4.4 dargestellt ist. Die Eingabe der dritten Iteration beinhaltet zwei Kanten, (D, G) und (D, H), die über Pfade der Länge 3 ($D{-}E{-}F{-}G$) bzw. 4 ($D{-}E{-}F{-}G{-}H$) konstruiert wurden. Bei der Kombination dieser beiden Pfade entsprechend der Grundidee würde ein Pfad (G, H) der Länge 7 resultieren. Allerdings existiert in der Ausgangsrelation R bereits ein Pfad (G, H) der Länge 1. Deswegen wird vorgeschlagen, in jeder Iteration den ersten und den letzten zu traversierenden Zwischenknoten mit dem Pfad zu speichern.

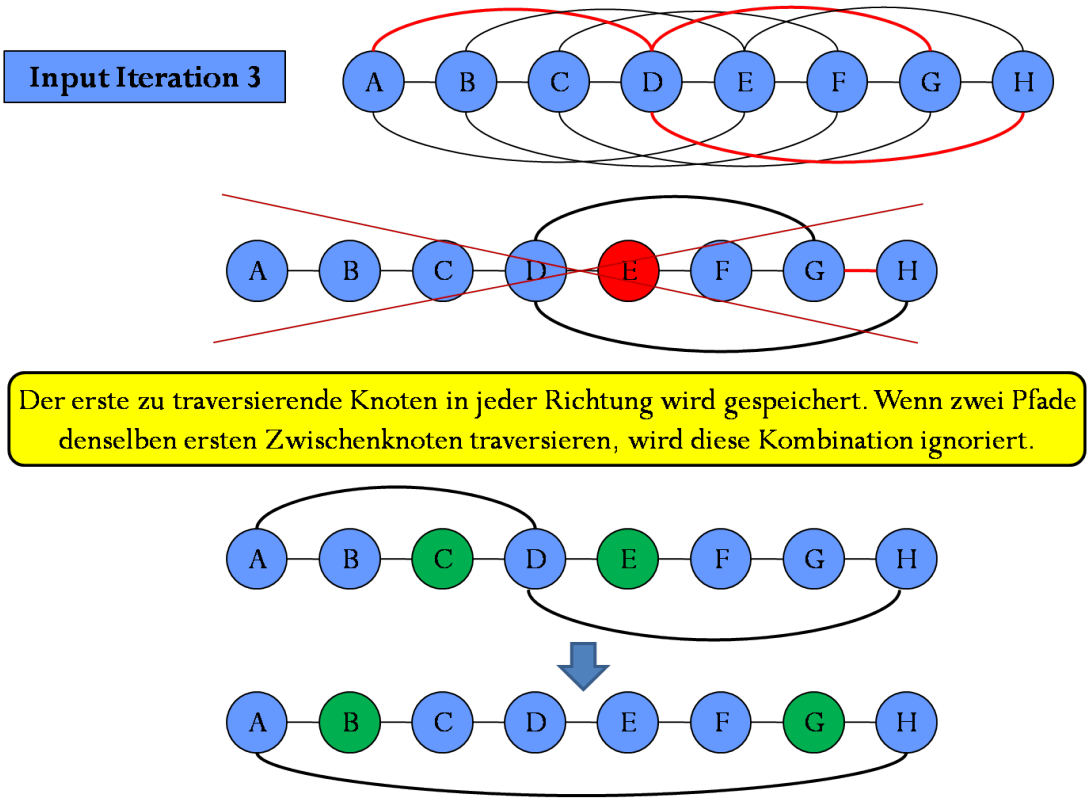

Abbildung 4.4: Direction-Problem und seine Lösung

Wenn zwei zu kombinierende Kanten denselben zu traversierenden Knoten haben, wird diese Kombination ignoriert. Im Beispiel der Abbildung 4.4 wird von D nach G der Knoten E zuerst traversiert, ebenso von D nach H. Weil (D, G) und (D, H) denselben ersten Knoten E aufweisen, werden sie *nicht* zu einem neuen Pfad kombiniert. Beim Pfad $D{-}C{-}B{-}A$ wird C als erster Knoten traversiert. Bei $D{-}E{-}F{-}G{-}H$ ist E der erste Zwischenknoten. Aus diesem Grund kann ein neuer Pfad von A nach G der Länge 7 generiert werden. Darüber hinaus werden für die nächsten Iterationen die Zwischenknoten B und G mit diesem Pfad gespeichert.

4.2 Algorithmen

Als Teil der Arbeit wurde Smart-MR entwickelt. Dies ist ein MapReduce-Algorithmus, welcher die transitive Hülle eines ungerichteten azyklischen Graphen berechnet. Er ist an den Smart-Algorithmus angelehnt, kommt jedoch ohne Union- oder Differenzoperationen für die Entfernung von Duplikaten aus. Stattdessen werden die zuvor genannten Strategien verwendet, um die Erzeugung von Duplikaten zu vermeiden. Für zyklische Graphen werden im Anschluss drei andere Verfahren (Cyc-Smart-MR, Full-TC-MR und CC-MR) vorgestellt.

4.2.1 Smart-MR

Smart-MR ist ein iterativer MapReduce-Algorithmus für die Berechnung der transitiven Hülle ungerichteter azyklischer Graphen. Er besteht aus zwei map-Funktionen und einer reduce-Funktion. In jeder Iteration (außer der ersten) erhält er zwei Relationen als Eingabe, die Ausgangsrelation und das Ergebnis der vorigen Iteration, und generiert eine Ausgabe, die wiederum mit der Ausgangsrelation als Eingabe in der nächsten Iteration verwendet wird. Die Ausführung von Smart-MR entspricht der in Abbildung 4.1 dargestellten Grundidee unter Beachtung der drei beschriebenen Kriterien, die die Generierung von Duplikaten verhindern.

map-Phase: Sie besteht aus zwei map-Funktionen, map_1 und map_2, die sich im Wesentlichen nur hinsichtlich der Signatur der Übergabeparameter unterscheiden (Abbildung 4.5). Entsprechend der Grundidee werden in dieser Phase aus jedem gelesenen Paar (x, y) zwei key-value-Paare ausgegeben, (x, y) und (y, x). Zur Behebung der drei analysierten Probleme wird der Value

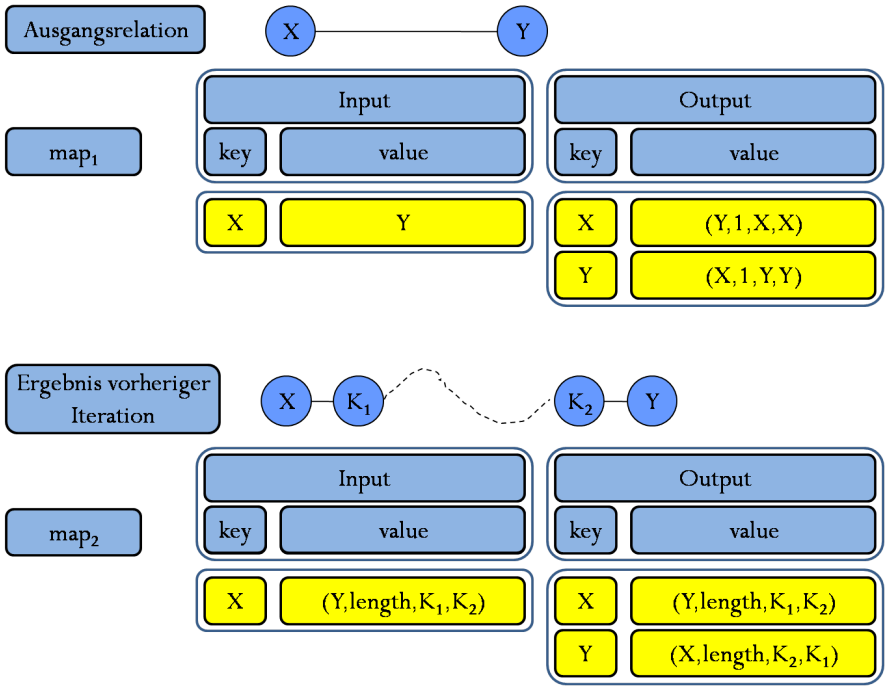

Abbildung 4.5: Datenformat der Ein- und Ausgabe der zwei map-Funktionen

jedes auszugebenden Paares um die Länge des Pfades und die zwei ersten in jede Richtung zu traversierenden Knoten angereichert. Die Funktion map_1 liest die Ausgangsrelation und generiert für jedes Eingabepaar (x, y) die zwei key-value-Paare $(x, (y, 1, x, x))$ und $(y, (x, 1, y, y))$. Die Funktion map_2 erhält Paare, die in der vorherigen Iteration ausgegeben wurden, i. e. der Value jedes Paares ist bereits mit der Länge und den Zwischenknoten angereichert. Deswegen liest die Funktion map_2 ein Eingabepaar der Form $(x, (y, length, k_1, k_2))$ und generiert daraus die zwei key-value-Paare $(x, (y, length, k_1, k_2))$ und $(y, (x, length, k_2, k_1))$.

Algorithm 4.1 Smart-MR (map)

// **map**$_1$ (Ausgangsrelation)
Input: (id_1, id_2)
Output: $(id_1, (id_2, 1, id_1, id_1))$ & $(id_2, (id_1, 1, id_2, id_2))$;

// **map**$_2$ (Ergebnis vorheriger Iteration)
Input: $(id_1, (id_2, length, inter_1, inter_2))$
Output: $(id_1, (id_2, length, inter_1, inter_2))$ & $(id_2, (id_1, length, inter_2, inter_1))$

Algorithm 4.2 Smart-MR (reduce)

```
1:  buffer = {};
2:  needNewIteration = false;
3:  reduce(key, Iterator<value>)
4:  while Iterator.hasNext() do
5:    v = Iterator.next();    /* v=(id, length, inter_1, inter_2) */
6:    for all e ∈ buffer do
7:      if (e.length < v.length) then
8:        continue;
9:      dist_{e,v} = e.length + v.length;
10:     if (|e.length − v.length| > 1 && dist_{e,v} ≠ 2) then
11:       continue;
12:     if (dist_{e,v} ⩽ 2^{iteration−1}) then
13:       continue;
14:     if (e.inter_1 = v.inter_1) then
15:       continue;
16:     Output (e.id, (v.id, dist_{e,v}, e.inter_2, v.inter_2));
17:     if (dist_{e,v} = 2^{iteration}) then
18:       needNewIteration = true;
19:   end
20:   buffer = buffer ∪ {v};
21: end
```

reduce-Phase: Die key-value-Paare der map-Phase werden umvereilt, sodass in der reduce-Phase alle Paare mit demselben Key beim selben reduce-Task gruppiert werden. Die reduce-Funktion des Framework Hadoop empfängt die gruppierten Values eines Keys als Datenfluss, der technisch mit einem Iterator bewerkstelligt wird (Zeile 3 im Algorithmus 4.2). Aus diesem Grund wird die Liste $buffer$ verwendet, die bereits gelesene Values speichert. Für jeden gelesenen Value aus dem $Iterator$ und jeden Value aus der Liste $buffer$ werden Paare $\{(x, y),\ x \in buffer\ \wedge\ y \in Iterator\}$ gebildet. Jedes Paar stellt zwei Pfade dar (vom key bis x und vom key bis y), die zu einem längeren Pfad kombiniert werden können. Von Zeile 7 bis Zeile 15 wird für das Paar überprüft, ob eines der drei Ausschlusskriterien vorliegt. Zeilen 7 bis 11 stellen fest, dass die zu kombinierenden Values einen Pfad darstellen, dessen Präfix kleiner als dessen Suffix ist, und dass der Längenunterschied zwischen ihnen nicht größer als 1 ist. Eine Ausnahme besteht trotzdem für die Pfade der Länge $2^{i-1} + 1$, die aus einem Präfix der Länge 1 und einem Suffix der Länge 2^{i-1} zusammengesetzt werden. Zeile 12 verhindert die Erstellung von kürzeren Pfaden (Sie wurden bereits in vorherigen Iterationen berechnet.). Schließlich

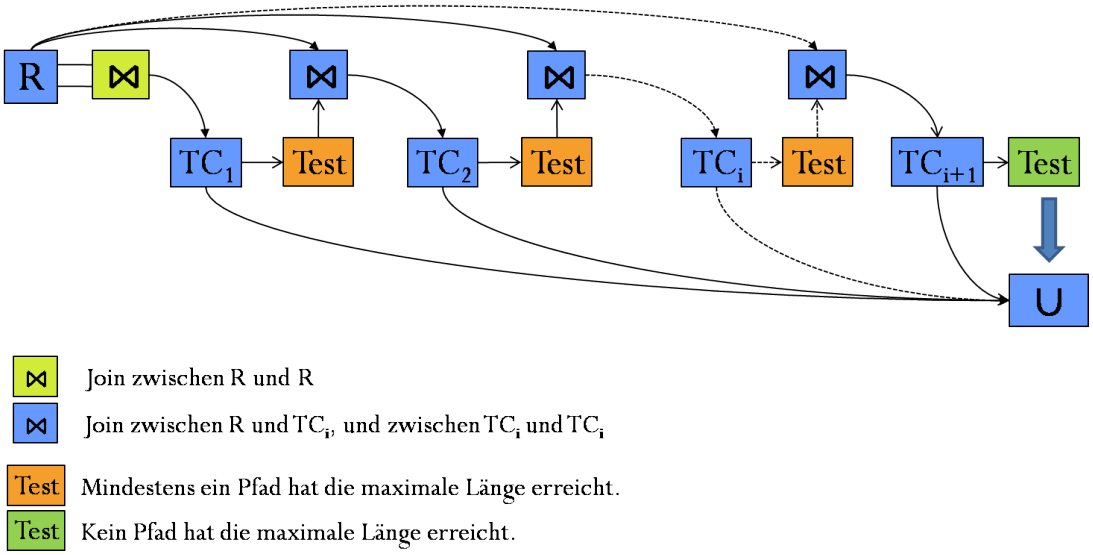

Abbildung 4.6: Der Ausführungsplan von Smart-MR

überprüft die Zeile 14, ob die Values denselben ersten Zwischenknoten besitzen. Wenn alle Ausschlusskriterien auf das Paar nicht zutreffen, wird ein neues Paar in Zeile 16 ausgegeben. Der Key von diesem Paar ist der Identifikator $e.id$ des aus $buffer$ gelesenen Elementes e. Der Value dieses Paares besteht aus dem Identifikator $v.id$ von $v \in Iterator$, aus der Summe der Längen von e und v und aus den zweiten Zwischenknoten $e.inter_2$ und $v.inter_2$. Die Längen von e und v, die summiert werden sollen, repräsentieren die Abstände ihrer jeweiligen Knoten zu dem mit key gekennzeichneten Knoten im Graphen. Die zwei Zwischenknoten $e.inter_2$ und $v.inter_2$ in der Ausgabe vom Value werden in der nächsten Iteration als $inter_1$ und $inter_2$ betrachtet, wobei der eine für den Test in Zeile 14 im Algorithmus 4.2 verwendet wird und der andere als Zwischenknoten in der Ausgabe eines neuen Pfades dient.

In jeder Iteration werden dieselben map- und reduce-Funktionen auf verschiedenen Knoten des MapReduce-Clusters ausgeführt. Der Ausführungsplan von Smart-MR ist in Abbildung 4.6 dargestellt, wobei die Union optional ist. Für die Terminierung des Algorithmus wird nicht verifiziert, ob neue Pfade generiert wurden oder nicht. Stattdessen wird in jeder Iteration die Zeile 25 des Algorithmus 4.2 ausgeführt. Sie überprüft, ob einer der erstellten Pfade die maximale Länge erreicht hat. Wenn dies der Fall ist, wird eine weitere Iteration durchgeführt, weil längere Pfade möglicherweise noch nicht berechnet wurden. Wenn im Gegenteil kein Pfad diese maximale Länge erreicht hat, hört die Berechnung auf, da alle Pfade erzeugt wurden und längere Pfade in den nächsten Iterationen nicht mehr entdeckt werden. Beispielsweise wird in der Iteration 4 geprüft, ob irgendein Pfad die Länge $2^4 = 16$ hat. Falls alle ent-

deckten Pfade eine Länge < 16 haben, terminiert die Berechnung, weil es keine längeren Pfade mehr gibt. Weist aber mindestens ein Pfad die Länge 16 auf, wird eine weitere Iteration durchgeführt, da Pfade, deren Länge > 16 ist, diese weitere Iteration benötigen. Eine zusätzliche *unnötige* Iteration wird allerdings durchgeführt, wenn die Tiefe d des Graphen genau 2^i ist, i. e. wenn im vorherigen Beispiel $d = 16$ ist, wird eine weitere Iteration (die fünfte) durchgeführt, die keine weiteren Pfade entdeckt. Damit konvergiert Smart-MR nach maximal $(\log d)+1$ Iterationen. Mit der folgenden Anpassung des Smart-MR-Algorithmus kann auf diese Iteration verzichtet werden, sodass der Algorithmus nach genau $\log d$ terminiert. In einem azyklischen ungerichteten Graphen ist der längste Pfad dadurch gekennzeichnet, dass sein Start- sowie sein Endknoten eine einzige Verbindung mit den anderen Knoten haben. Diese Eigenschaft kann in Smart-MR ausgenutzt werden, indem Pfade mit oben beschriebenen Start- und Endknoten markiert werden. Wenn in Iteration i ein Pfad die maximale Länge 2^i erreicht, wird überprüft, ob mindestens einer seiner Start- und Endknoten zwei Verbindungen hat. Trifft dies zu, deutet das auf die Existenz längerer Pfade (Länge $> 2^i$) hin, die mit einer weiteren Iteration entdeckt werden. Gibt es aber keinen solchen Pfad, so wird auch keine weitere Iteration durchgeführt und Smart-MR kann nach genau $\log d$ konvergieren.

Wie schon erwähnt, wurde Smart-MR für die Berechnung der transitiven Hülle azyklischer Graphen implementiert. Duplikate werden dabei gar nicht generiert. Zyklische Graphen erzeugen allerdings Duplikate, die Smart-MR meistens nicht als solche erkennt. Das hat zur Folge, dass Smart-MR entweder mehr Iterationen benötigt oder gar nicht terminiert. In den folgenden Abschnitten werden für die Handhabung des Problems andere Lösungen präsentiert.

4.2.2 Cyc-Smart-MR

In zyklischen Graphen kann von einem Knoten aus ein anderer Knoten über mehrere Pfade erreicht werden. Dabei ist nur der kürzeste Pfad von Bedeutung. Alle anderen Pfade sind bei der Berechnung der transitiven Hülle reine Duplikate mit größeren Pfadlängen. Wie im Abschnitt 4.1.1 bereits erläutert wurde, führen Duplikate einerseits zu einem exponentiellen Zuwachs der Datenmenge und andererseits dazu, dass die Berechnung gar nicht terminiert. Wenn Smart-MR die transitive Hülle des Graphen der Abbildung 4.7 berechnet, entdeckt er bereits in der ersten Iteration den Pfad mit der Länge 2 von A nach J über H. In der dritten Iteration seiner Ausführung entdeckt Smart-MR durch die Zusammensetzung der Pfade $E{-}D{-}C{-}B{-}A$ und $E{-}F{-}G{-}H{-}J$ wieder den Pfad von A nach J, jedoch mit der Länge 8. Dass es sich hierbei um ein Duplikat handelt, kann aus folgendem Grund nicht erkannt werden: In einer

Iteration i kennt Smart-MR nur das Ergebnis der Iteration $i-1$ (und natürlich die Ausgangsrelation) und hat keinen Zugriff auf das Ergebnis der Iteration j, wenn $j < i-1$. Der längere Pfad von A nach J wird in der dritten Iteration entdeckt, die ausschließlich das Ergebnis der zweiten Iteration kennt. Da der kürzere Pfad $A-H-J$ aber bereits in der ersten Iteration erzeugt wurde, gibt es keine Möglichkeit, die Daten miteinander zu vergleichen.

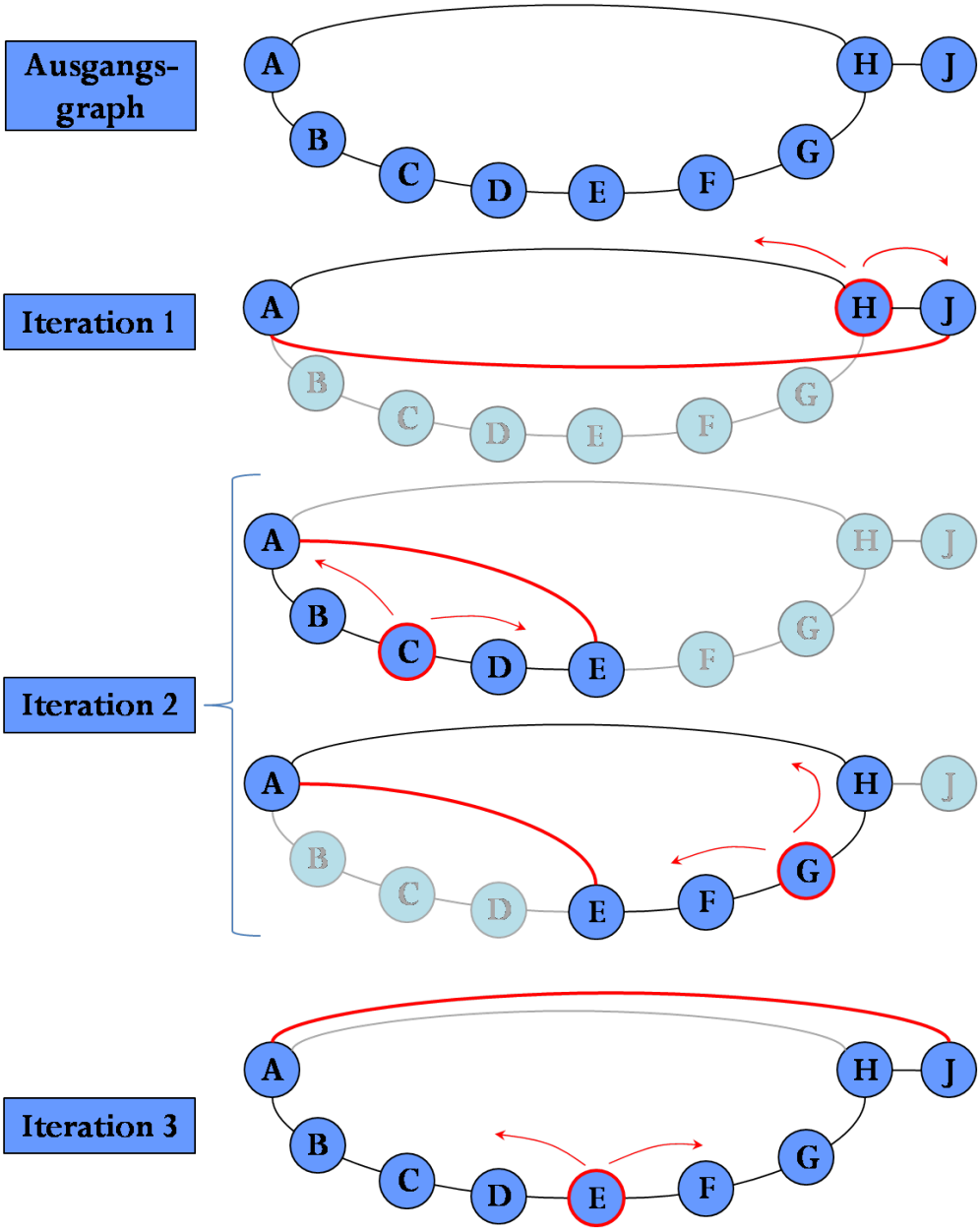

Abbildung 4.7: Beispiel eines zyklischen Graphen und das Problem der Entstehung von Duplikaten in verschiedenen Iterationen

Es kann festgehalten werden, dass die dritte Iteration völlig überflüssig ist, da die Länge des kürzesten Pfades zwischen zwei Knoten im Graphen maximal 4 beträgt und dieser bereits in der Iteration 2 entdeckt wurde. Mit anderen Worten, die Berechnung der transitiven Hülle eines zyklischen Graphen mit Smart-MR führt dazu, dass der Algorithmus gar nicht terminiert, weil immer wieder Duplikate mit falschen Pfadlängen generiert werden.

Cyc-Smart-MR ist eine Variante von Smart-MR für die Entdeckung von Duplikaten, die durch zyklische Graphen entstehen können. Er verhindert, dass diese zur Generierung von neuen Pfaden beitragen. Die Grundidee dafür ist die Speicherung aller möglichen Zwischenknoten, die von einem Quellknoten zu einem Zielknoten führen können. Wenn sich zwei Pfade an einem oder mehreren Knoten überschneiden, werden sie nicht zu einem längeren Pfad kombiniert. Im Folgenden werden nur die Abweichungen von Cyc-Smart-MR gegenüber Smart-MR erläutert.

map-Phase: Cyc-Smart-MR hat zwei map-Funktionen, die sich genauso wie die von Smart-MR verhalten, i. e. aus einem gelesenen Paar (x, y) generieren sie zwei key-value-Paare, (x, y) und (y, x). Darüber hinaus wird der Value eines key-value-Paares mit der Länge des Pfades und der Menge der traversierten Knoten angereichert.

Algorithm 4.3 Cyc-Smart-MR (map)

// **map**$_1$ (Ausgangsrelation)
Input: (id_1, id_2)
Output: $(id_1, (id_2, 1, \{id_2\}))$ & $(id_2, (id_1, 1, \{id_1\}))$

// **map**$_2$ (Ergebnis vorheriger Iteration)
Input: $(id_1, (id_2, length, Set<inter>)$
Output: $(id_1, (id_2, length, Set<inter>)$ & $(id_2, (id_1, length, Set<inter>)$

reduce-Phase: In dieser Phase werden, wie gewöhnlich in MapReduce, alle Values mit demselben Key bei derselben reduce-Funktion gruppiert, allerdings mit einer speziellen Sortierung. Dank *Secondary sort* von Hadoop liefert der Iterator den Value v_1 vor dem Value v_2, wenn $v_1.id < v_2.id$. Ist aber $v_1.id = v_2.id$, was auf ein Duplikat hindeutet, wird v_1 nur dann gelesen und im $buffer$ gespeichert, wenn $v_1.length < v_2.length$. In diesem Falle wird v_2 ignoriert und seine Zwischenknoten zu denen von v_1 hinzugefügt (Zeile 8). Dieser Prozess wird fortgesetzt und alle Values werden nach ids sortiert in der duplikatenfreien Liste $buffer$ gespeichert. Als nächstes werden aus Values der Liste Paare $\{(x, y),\ x, y \in buffer\ \wedge\ x < y\}$ gebildet und in den Zeilen 15 bis 23 analog zu

Smart-MR auf die Ausschlusskriterien geprüft. Der einzige Unterschied dabei ist die Zeile 22, in der nach Überschneidung der Zwischenknoten beider Values gesucht wird. Ist das der Fall, werden diese Pfade aufgrund der Überlappung nicht zu einem längeren Pfad kombiniert. Treffen die Ausschlusskriterien auf ein Paar (x, y) nicht zu, wird in Zeile 24 das Paar $(key_{out}, value_{out})$ ausgegeben, dessen Key key_{out} die id des ersten Values x ist, und dessen Value $value_{out}$ aus der id des zweiten Values y, der Summe der Längen beider Values x und y sowie der Union ihrer Zwischenknoten besteht.

Algorithm 4.4 Cyc-Smart-MR (reduce)

```
 1: buffer = []    // Sortierte Liste von Values
 2: needNewIteration = false;
 3: last = null;
 4: reduce(key, Iterator<value>)
 5: while Iterator.hasNext() do
 6:    v = Iterator.next();    /* (id, length, inter) */
 7:    if (last ≠ null && v.id = last.id) then
 8:       last.inter = last.inter ∪ v.inter;
 9:    else
10:       last = v;
11:       buffer = buffer ∪ {v};
12: end
13: for all (e ∈ buffer) do
14:    for all (v ∈ buffer) do
15:       if (e.length < v.length) then
16:          continue;
17:       dist_{e,v} = e.length + v.length;
18:       if (|e.length − v.length| > 1 && dist_{e,v} ≠ 2) then
19:          continue;
20:       if (dist_{e,v} ⩽ 2^{iteration−1}) then
21:          continue;
22:       if (e.inter ∩ v.inter ≠ ∅) then
23:          continue;
24:       output(e.id, (v.id, dist_{e,v}, e.inter ∪ v.inter);
25:       if (dist_{e,v} = 2^{iteration}) then
26:          needNewIteration = true;
27:    end
28: end
```

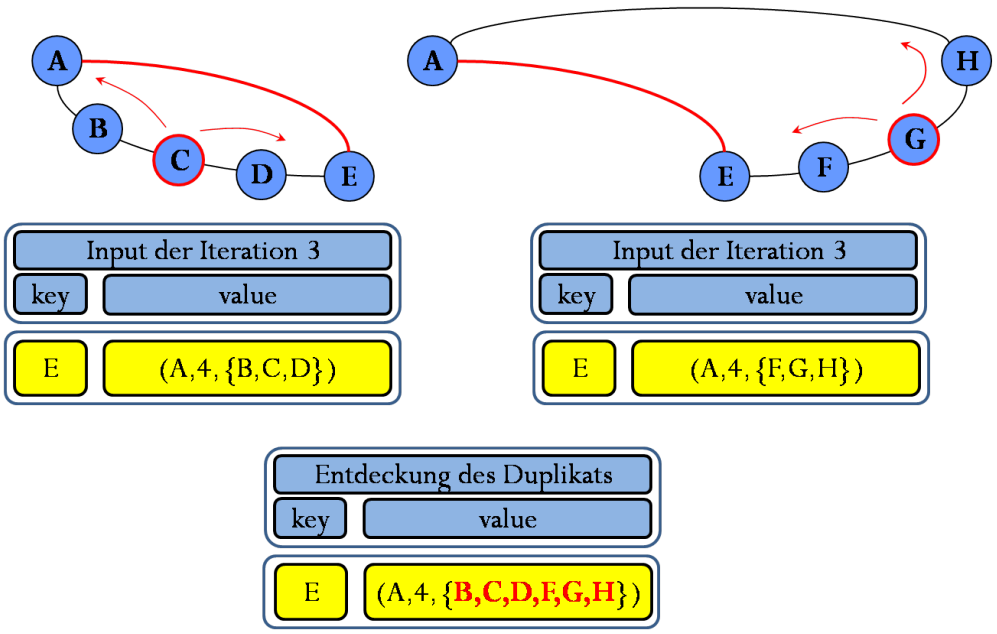

Abbildung 4.8: Die Entdeckung generierter Duplikate eines zyklischen Graphen und ihre Vereinigung zu einem einzigen Paar in Cyc-Smart-MR

Das Verhalten von Cyc-Smart-MR kann mit dem Graphen der Abbildung 4.7 verdeutlicht werden. In der ersten Iteration wird u. a. der Pfad von A nach J über H mit der Länge 2 entdeckt. In der zweiten Iteration werden schon die ersten Duplikate erzeugt, so wird z. B. der Pfad von A nach E zweimal ausgegeben, einmal über $B{-}C{-}D$ und einmal über $H{-}G{-}F$. Solche Duplikate werden in dieser zweiten Iteration nicht entdeckt, weil sie verschiedene Keys haben (C bzw. G), die möglicherweise in verschiedenen reduce-Funktionen berechnet wurden. Erst in der dritten Iteration werden sie als Duplikat erkannt. Der Key E hat u. a. diese drei Values: $\{(A,4,\{F,G,H\}),\ (A,4,\{B,C,D\}),\ (J,4,\{F,G,H\})\}$. Der erste Value $(A,4,\{B,C,D\})$ wird gelesen und in $buffer$ gespeichert. Der nächste Value $(A,4,\{F,G,H\})$ wird als Duplikat erkannt und nicht zu $buffer$ hinzugefügt. Stattdessen wird die Menge seiner Zwischenknoten $\{F,G,H\}$ zur Menge $\{B,C,D\}$ vom ersten gelesenen Value hinzugefügt. Dieser Prozess ist in Abbildung 4.8 dargestellt und sein Ergebnis ist der in $buffer$ gespeicherte Value $(A,4,\{B,C,D,F,G,H\})$. Beim Lesen des dritten Value $(J,4,\{F,G,H\})$ wird versucht, einen neuen Pfad von A nach J mit der Länge 8 zu generieren. Da sich die Mengen der Zwischenknoten von A und J überschneiden, wird die Kombination verworfen.
Das Abbruchkriterium in Cyc-Smart-MR ist dasselbe wie in Smart-MR. Am Ende jeder Iteration wird überprüft, ob mindestens einer der neu generierten Pfade

die maximale Pfadlänge erreicht hat. Cyc-Smart-MR konvergiert in der Regel nach $\log d$ Iterationen, in seltenen Fällen aber auch nach $(\log d) + 1$ Iterationen (falls die Tiefe des Graphen genau $2, 4, 8, 16, ..., 2^i$ ist).

Cyc-Smart-MR hat jedoch zwei Nachteile. Der erste ist, dass er die Duplikate erkennt, aber nicht entfernt. In einer Iteration i erkennt er die Duplikate, die in Iteration $i - 1$ generiert wurden, und verhindert nur, dass sie weitere Pfade erzeugen. Er kann sie nicht entfernen, da sie in das Dateisystem geschrieben wurden. Demzufolge muss am Ende der Berechnung der transitiven Hülle ein weiterer MapReduce-Job gestartet werden, der die generierten Duplikate aller Iterationen entfernt. Der noch größere Nachteil dieses Algorithmus ist jedoch der benötigte Speicherbedarf für die Menge aller Zwischenknoten eines Pfades. Auswertungen zeigten, dass die Ausführung von Cyc-Smart-MR für große dichte Graphen nach ein paar Iterationen abbricht.

4.2.3 Full-TC-MR

Für die Entdeckung von Duplikaten, die in der vorherigen Iteration generiert wurden, speichert Cyc-Smart-MR alle Zwischenknoten, die von einem Quellknoten zu einem Zielknoten traversiert werden können, und überprüft, ob sich die Mengen der Zwischenknoten von zwei zu kombinierenden Pfaden überschneiden. Diese Lösung funktioniert für dünne Graphen, weist jedoch ein erhebliches Skalierbarkeitsproblem für dichte Graphen auf. Eine andere Möglichkeit für die Erkennung von Duplikaten ist die Verwendung *aller Ergebnisse der vorherigen Iterationen*.
Wie die Abbildung 4.9 zeigt, verwendet Full-TC-MR eine einzige Relation als Eingabe. Diese beinhaltet in der Iteration i die Ergebnisse aller Iterationen $j < i$. Nach der Berechnung der transitiven Hülle in jeder Iteration wird getestet, ob in dem Ergebnis mindestens ein Paar die maximale Länge erreicht hat. Ist das der Fall, wird die Berechnung der transitiven Hülle fortgesetzt, da es möglicherweise längere Pfade gibt, die noch nicht entdeckt worden sind. Wenn aber kein Pfad die maximale Länge erreicht hat, wird die Berechnung abgebrochen und ein weiterer MapReduce-Job ausgeführt, der die Duplikate der letzten Iteration entfernt.

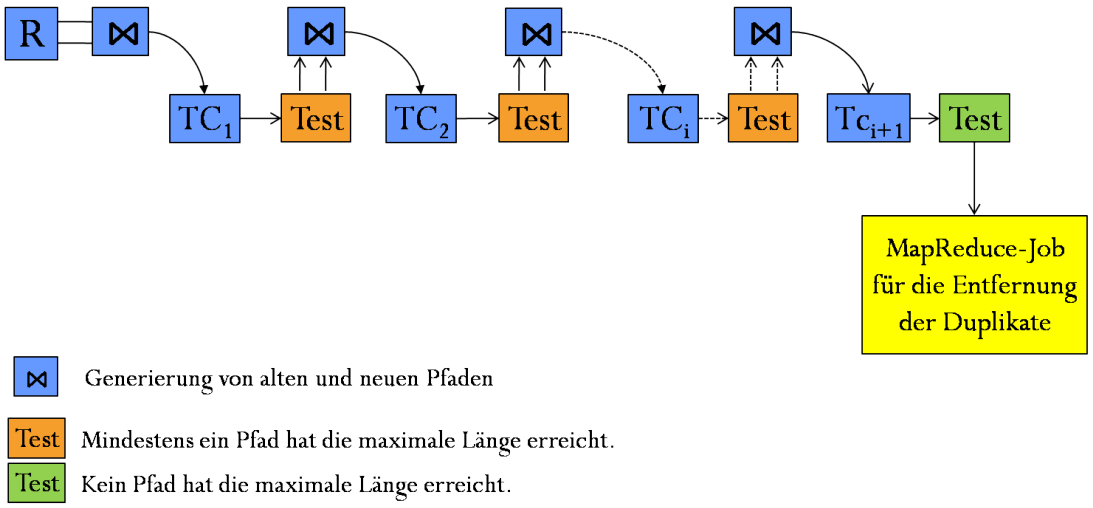

Abbildung 4.9: Der Ausführungsplan von Full-TC-MR

map-Phase: Full-TC-MR hat nur eine map-Funktion. Sie liest in der ersten Iteration ein Paar (x, y) aus der Ausgangsrelation und erzeugt zwei key-value-Paare $(x, (y, 1))$ und $(y(x, 1))$, wobei der Value jedes der beiden Paare aus dem Quellknoten und der Länge 1 eines Pfades besteht. Da in der nächsten Iteration längere Pfade entdeckt werden, werden Values der Paare mit den richtigen Längen angereichert, i. e. die map-Funktion liest das Paar $(x, (y, length))$ und gibt $(x, (y, length))$ und $(y(x, length))$ für die reduce-Funktion aus.

Algorithm 4.5 Full-TC-MR (map)

Input: $(id_1, (id_2, length))$
Output: $(id_1, (id_2, length))$ & $(id_2, (id_1, length))$

reduce-Phase Mit der Verwendung von *Secondary sort* werden die Values, die bei einem Key gruppiert sind, sortiert eingelesen. Sie werden wie bei Cyc-Smart-MR nach ids aufsteigend sortiert. Sind die ids zweier Values gleich, werden sie nach Pfadlängen wieder aufsteigend sortiert. Diese Sortierung ermöglicht die Entdeckung von Duplikaten, da deren Values nacheinander eingelesen werden. Das geschieht in Zeile 7 des Algorithmus, wo im Falle von Duplikaten nur der erste Value mit der kürzesten Länge dank der Sortierung akzeptiert wird und alle anderen verworfen werden. Danach wird jeder akzeptierte Value für zwei Aufgaben verwendet, einmal für die Erzeugung von neuen Pfaden (Zeile 9 bis 19), und einmal für die Generierung von alten Pfaden, die in den vorherigen Iterationen erzeugt wurden (Zeilen 21 und 22). In der

Schleife *for* wird der zuletzt gelesene Value mit den Values der Liste $buffer$ als Paar auf die Ausschlusskriterien in Zeilen 10 bis 16 getestet. Da wegen der Skalierbarkeit die traversierten Knoten eines Pfades nicht gespeichert werden, wird es in Full-TC-MR keinen Test hinsichtlich gemeinsamer Zwischenknoten von zwei Values geben. Das führt dazu, dass nach dieser reduce-Phase Duplikate generiert werden können, die erst in der nächsten Iteration erkannt und entfernt werden. Treffen die Kriterien auf das Paar nicht zu, wird es mit der neuen Länge in Zeile 17 ausgegeben. Außerdem werden gültige kürzere Pfade, die in den vorherigen Iterationen generiert wurden, noch einmal in Zeile 22 ausgegeben. Dabei werden zur Vermeidung von Duplikaten nur Paare generiert, deren Zielknoten (oder Value) größer als der Key ist.

Algorithm 4.6 Full-TC-MR (reduce)

```
 1: buffer = []    // Sortierte Liste von Values
 2: needNewIteration = false;
 3: last = null;
 4: reduce(key, Iterator<value>)
 5: while (Iterator.hasNext()) do
 6:    v = Iterator.next();    /* (id, length, inter) */
 7:    if (last ≠ null  &&  v.id = last.id) then
 8:       continue;
 9:    for all (e ∈ buffer) do
10:       if (e.length < v.length) then
11:          continue;
12:       dist_{e,v} = e.length + v.length;
13:       if (|e.length − v.length| > 1 && dist_{e,v} ≠ 2) then
14:          continue;
15:       if (dist_{e,v} ⩽ 2^{iteration−1}) then
16:          continue;
17:       output(e.id, (v.id, dist_{e,v}));
18:       if (dist_{e,v} = 2^{iteration}) then
19:          needNewIteration = true;
20:    end
21:    if (key < v.id) then
22:       output(key, (v.id, v.length));
23:    last = v;
24: end
```

Full-TC-MR konvergiert nach maximal $(\log d) + 2$ Iterationen. Außerdem muss ein anderer MapReduce-Job am Ende der Berechnung der transitiven Hülle ausgeführt werden, der die Duplikate der letzten Iteration entfernt.

4.2.4 CC-MR

CC-MR ist eine neuer Algorithmus [SBF12], der zusammenhängende Komponenten eines Graphen entdeckt und aus diesen Spannbäume konstruiert. Dafür setzt er eine Ordnung „$<$ “ der Knoten des Graphen voraus. Mittels dieser Ordnung überprüft der Algorithmus, ob alle Nachbarn eines Knoten x größer als er selbst sind oder nicht. Ist das der Fall, verbindet CC-MR all die Nachbarn mit x. Andernfalls findet er y, den kleinsten Nachbarn von x, und ordnet all diese Nachbarn inklusive x dem kleinsten Knoten y zu. Die Fortsetzung dieses Prozesses generiert die gewünschten zusammenhängenden Komponenten eines Graphen, wobei jede Komponente mit ihrem kleinsten Knoten identifiziert wird und ihre anderen Knoten nur mit diesem kleinsten Knoten verbunden sind. Um das zu realisieren, modifiziert CC-MR den originalen Graphen, indem Kanten gelöscht und andere hinzugefügt werden. Zwei Arten von gerichteten Kanten werden hinzugefügt: *Vor-* und *Rückwärtskanten*. Eine gerichtete Kante $x \to y$ wird als Vorwärtskante bezeichnet, wenn $x < y$. Ist jedoch $x > y$, so handelt es sich um eine Rückwärtskante. Diese Art von Kanten (Rückwärtskanten) dienen der Verbindung des kleineren Knoten y mit anderen Knoten im Graphen, die nur über den größeren Knoten x erreichbar sind. Der Algorithmus CC-MR repräsentiert jede Kante $x \to y$ mit einem key-value-Paar (x, y), wobei die Feststellung, dass es sich um eine Vor- oder Rückwärtskante handelt, durch den Vergleich beider Knoten miteinander erfolgt. Dieselbe key-value-Struktur wird in der map- und reduce-Phase verwendet.

map-Phase: In der ersten Iteration erzeugt die map-Funktion aus jeder Kante (x, y) des ungerichteten Graphen zwei gerichtete Kanten, (x, y) und (y, x). Dies geschieht, weil die reduce-Funktion von CC-MR gerichtete Kanten benötigt. In den folgenden Iterationen verhält sich die map-Funktion jedoch wie ein *identity mapper*, i. e. sie gibt jedes gelesene Paar ohne Änderung aus. Dieses Verhalten liegt darin begründet, dass die map-Funktion einer Iteration $j > 1$ das Ergebnis der reduce-Phase von der Iteration $j - 1$ liest, das bereits gerichtete Kanten darstellt.

Algorithm 4.7 CC-MR (map)

// Für die erste Iteration
Input: (id_1, id_2)
Output: (id_1, id_2) & (id_2, id_1)
// Für alle anderen Iterationen
Input: (id_1, id_2)
Output: (id_1, id_2)

Algorithm 4.8 CC-MR (reduce)

```
1: newIterationNeeded= false;
2: reduce(source, Iterator<value>)
3: LocalMax= false;
4: first = Iterator.next();
5: if source < first then
6:    LocalMax = true;
7:    output(source, first);
8: dest_old = first;
9: while (Iterator.hasNext()) do
10:   dest = it.next();
11:   if dest = dest_old then
12:      continue;    // remove duplicates
13:   if LocalMax then
14:      output(source, dest);    // only forward edge
15:   else
16:      output(first, dest);    // forward edge
17:      output(dest, first);    // backward edge
18:      newIterationNeeded= true;
19:   dest_old = dest;
20: if (source < dest_old && !LocalMax) then
21:   output(source, first);    // backward edge
```

reduce-Phase: Jede reduce-Funktion empfängt die entsprechenden Values eines Keys gruppiert und aufsteigend sortiert. Am Anfang wird angenommen, dass der Key $source$ das kleinste Element und somit der Bezeichner dieser zusammenhängenden Komponente ist. Der entscheidende Vergleich geschieht in Zeile 5, wo $source$ mit dem ersten Value $first$ aus $Iterator$ verglichen wird. Der Fall $source < first$ bedeutet, dass der Key (bis zu dieser Iteration) tatsächlich das kleinste Element und der Bezeichner dieser Komponente ist. Darauf folgen die Ausgaben in Zeilen 7 und 14 vom Key $source$ mit allen Values $\{v, v \in Iterator\}$ als Paare $(source, v)$, die Vorwärtskanten darstellen. Wenn aber $source > first$, bedeutet dies, dass nicht $source$ der tatsächliche Bezeichner dieser Komponente ist, sondern $first$. Aus diesem Grund werden alle anderen Values $\{v, v \in Iterator\}$ mit dem Value $first$ zunächst als Paare $(first, v)$, die Vorwärtskanten darstellen, in Zeile 16 ausgegeben. Zusätzlich zu den Vorwärtskanten werden Rückwärtskanten $(v, first)$ ausgegeben, die zur Entdeckung neuer Knoten in den nächsten Iterationen verwendet werden. Außerdem wird in Zeile 20 überprüft, ob der Key $source$ das größte Element

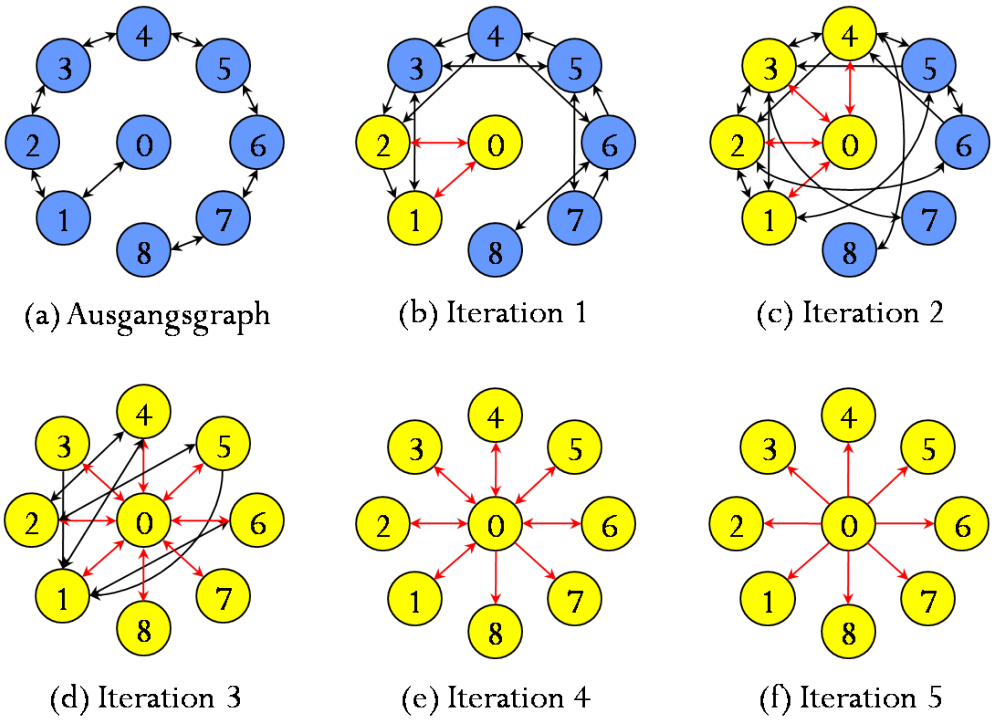

Abbildung 4.10: Die Entdeckung der zusammenhängenden Komponente eines Graphen durch die Ausführung von CC-MR in mehreren Iterationen

innerhalb dieser Komponente ist. Falls er weder das größte noch das kleinste ist, wird eine Rückwärtskante mit dem Paar $(source, first)$ ausgegeben. Diese Rückwärtskante hat die gleiche Bedeutung wie die anderen Rückwärtskanten. Solange der Algorithmus Rückwärtskanten generiert, werden weitere Iterationen benötigt. CC-MR terminiert in der Regel nach $(\log d) + k$ Iterationen. In $\log d$ werden alle Knoten einer zusammenhängenden Komponente mit den kleinsten Knoten unter ihnen verbunden. Die zusätzlichen k Iterationen dienen der Entfernung der verbleibenden Rückwärtskanten.

Die Idee der Vorwärts- und Rückwärtskanten, die CC-MR für die Entdeckung von zusammenhängenden Komponenten verwendet, wird anhand des Beispiels in Abbildung 4.10 erläutert. In der ersten Iteration wird in einer reduce-Funktion der Knoten 1 als Key ($source$) betrachtet, dessen Values die zwei Knoten 0 und 2 sind. Da aber $1 > 0$ ist (der Fall $source > first$), werden die Knoten 0 und 2 durch eine Vorwärts- und Rückwärtskante miteinander verbunden. Außerdem wird entsprechend Zeile 20 des Algorithmus der Knoten 1 ($source$) mit 2 durch eine Rückwärtskante verknüpft. In der zweiten Iteration erfolgt aufgrund der in der vorherigen Iteration erzeugten Vorwärtskanten $1 \rightarrow 3$ und $2 \rightarrow 4$ sowie der Rückwärtskanten $1 \rightarrow 0$ und $2 \rightarrow 0$ eine Verbindung

der Knoten 3 und 4 mit dem kleinsten Knoten 0. Bereits in der dritten Iteration werden alle Knoten mit dem kleinsten, i. e. Knoten 0, in einem sternähnlichen Schema verbunden, allerdings werden zwei weitere Iterationen (Iteration 4 und 5) benötigt, um die Rückwärtskanten zu entfernen.

Die Anpassung von CC-MR für die Berechnung der transitiven Hülle ist recht einfach. Am Ende der Ausführung von CC-MR sehen die Verbindungen der Knoten jeder zusammenhängenden Komponente cc im Graphen wie folgt aus: $\{(x, y),\ x, y \in cc \ \wedge\ \forall y \in cc \setminus x :\ x < y\}$. Beispielsweise ist die Menge der gerichteten Kanten $\{(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(0,7),(0,8)\}$ das Ergebnis der Ausführung von CC-MR auf dem Graphen der Abbildung 4.10(a). Ein zusätzlicher MapReduce-Job benutzt die von CC-MR erzeugten Kanten als Eingabe. In seiner map-Phase verwendet er einen *identity mapper*, der ein gelesenes key-value-Paar ohne Änderung ausgibt. In der reduce-Phase dieses Jobs werden alle Knoten einer zusammenhängenden Komponente bei derselben reduce-Task gruppiert, wobei der Key der kleinste Knoten ist und die Values die anderen Knoten bilden. Ein kartesisches Produkt von diesen Values generiert Paare, die mit der Eingaberelation (Ergebnis von CC-MR) die transitive Hülle des originalen Graphen darstellen.

4.2.5 Kritik und mögliche Verbesserung von CC-MR

Obwohl die Auswertungen generell eine Überlegenheit von CC-MR gegenüber den anderen Algorithmen zeigen, könnte seine Grundidee einen Nachteil bei manchen dichten Graphen aufweisen. CC-MR setzt, wie schon beschrieben, eine Ordnung der Ids der Knoten voraus, i. e. die Ids der Knoten können numerisch oder lexikografisch aufsteigend sortiert werden. Innerhalb einer zusammenhängenden Komponente werden iterativ gerichtete Kanten gelöscht und andere Vor- und Rückwärtskanten hinzugefügt, sodass am Ende alle Knoten einer zusammenhängenden Komponente nur mit jenem Knoten verbunden sind, der die kleinste Id hat. Diese Vorgehensweise kann aber dazu führen, dass viele Rückwärtskanten generiert werden, die unnötige Iterationen verursachen. Um das Verhalten zu erläutern, betrachten wir das Beispiel in Abbildung 4.11. Der Ausgangsgraph besteht aus einer einzigen zusammenhängenden Komponente, innerhalb derer der Knoten 0 die kleinste Id besitzt. Gemäß dem Algorithmus CC-MR werden alle anderen Knoten mit den Ids $1, 2, 3, 4, 5, 6, 7$ und 8 iterativ mit dem Knoten 0 verbunden. Obwohl dies bereits in der zweiten Iteration geschieht, wird eine dritte Iteration benötigt, um die Rückwärtskanten $1 \to 0$, $2 \to 0$ und $6 \to 0$ zu entfernen.

Eine Alternative zur numerischen oder lexikographischen Sortierung der Ids der Knoten wäre die Ordnung der Knoten nach ihren Graden. In der Graphen-

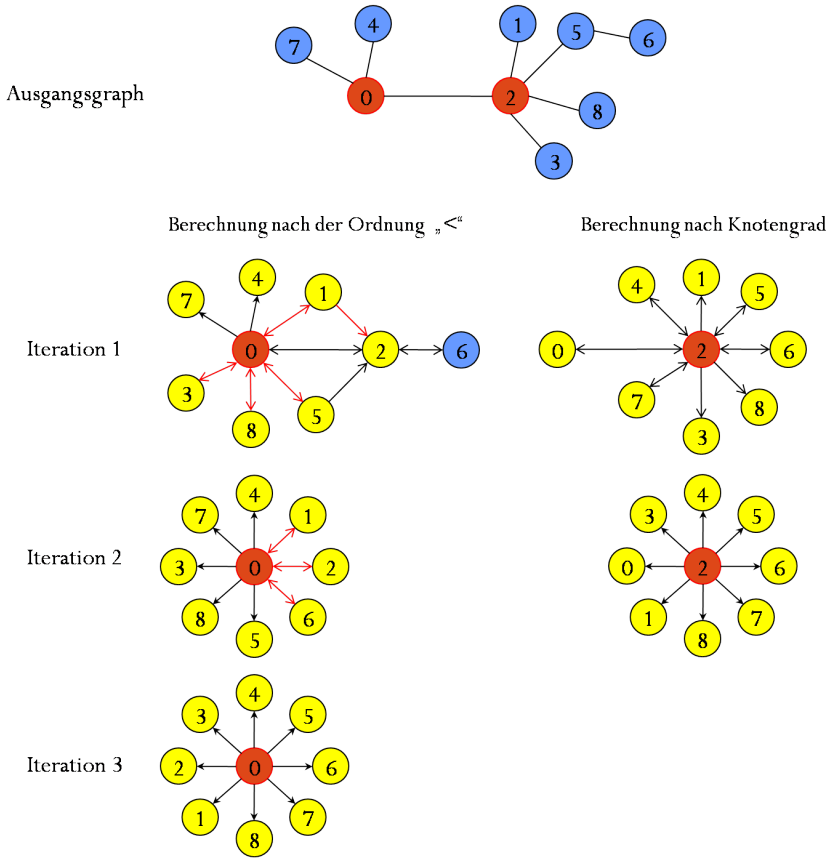

Abbildung 4.11: Zwei Methoden für die Entdeckung von zusammenhängenden Komponenten eines Graphen, links wenn die Knoten nach Ids und rechts wenn sie nach ihren Graden sortiert werden

theorie bezeichnet der Grad eines Knoten v im Graphen $G(V, E)$ die Anzahl der Nachbarn $N(v) = \{u, \in V \mid (v, u) \in E\}$ von v, i. e. $deg(v) = |N(v)|$. Bei gerichteten Graphen wird zwischen dem Ausgangsgrad $deg^+(v)$ und dem Eingangsgrad $deg^-(v)$ unterschieden, die jeweils die ausgehenden bzw. eingehenden Kanten eines Knoten v darstellen. Bei ungerichteten Graphen hingegen wird jede Verbindung zwischen zwei Knoten gleichzeitig als Vor- und Rückwärtskante betrachtet. Beispielsweise wird die Kante $0 - 2$ der Abbildung 4.11 für den Knoten 0 einmal als ausgehende und einmal als eingehende Kante angesehen. Das Gleiche gilt für Knoten 2. Im Folgenden wird nur die Anzahl der ausgehenden Kanten, also deg^+, eines Knoten betrachtet. In diesem Kontext heißt eine Kante $x \rightarrow y$ Vorwärtskante, wenn $deg^+(x) > deg^+(y)$, andernfalls ist sie eine Rückwärtskante (Im Falle von Gradgleichheit werden einfach die Ids der Knoten wie in CC-MR verglichen.).
CC-MR könnte so geändert werden, dass die Knoten einer zusammenhängen-

den Komponente nicht mit dem kleinsten Knoten verbunden werden, sondern mit dem Knoten, der den größten deg^+ hat. Dies könnte im Vergleich zur originalen Version von CC-MR eine in zweierlei Hinsicht positive Wirkung bei dichten Graphen haben. Einerseits führt die Änderung zu einer geringeren Generierung von Rückwärtskanten und damit auch zu weniger Daten, die in das Dateisystem HDFS geschrieben und anschließend im Cluster umverteilt werden müssen. Dafür gibt es die folgende Erklärung: Am Anfang sind die Verbindungen zwischen den Knoten des ungerichteten Graphen jeweils durch eine Vor- und eine Rückwärtskante repräsentiert. Da der Knoten mit den größten deg^+ mit mehr Knoten verbunden ist als die anderen, werden in den folgenden Iterationen weniger Vor- und Rückwärtskanten benötigt, um ihn mit neuen Knoten zu verbinden. Andererseits kann die geringe Erzeugung von Rückwärtskanten ein schnelleres Konvergieren des Algorithmus ermöglichen, da die Terminierung bei CC-MR eine Nicht-Generierung und das Entfernen aller Rückwärtskanten voraussetzt. Im rechten Beispiel der Abbildung 4.11, der die neue Idee darstellt, besitzt der Knoten 2 den größten Grad, $deg^+(2) = 5$, und wird als Bezeichner der zusammenhängenden Komponente ausgewählt. Alle anderen Knoten $0, 1, 3, 4, 5, 6, 7$ und 8 werden iterativ mit dem Knoten 2 und nur mit ihm verbunden. Dieser wird in der ersten Iteration nur noch mit den drei neuen Knoten $4, 7$ und 6 durch jeweils eine Vor- und eine Rückwärtskante verlinkt werden. Außerdem entfallen in dieser Iteration die Rückwärtskanten $1 \to 2$, $3 \to 2$ und $8 \to 2$. In der zweiten und damit letzten Iteration werden alle Rückwärtskanten entfernt, da keine neuen Knoten mehr entdeckt werden. Im Gegenteil dazu werden entsprechend dem Algorithmus CC-MR in der ersten Iteration Vor- und Rückwärtskanten zwischen den Knoten $1, 3, 5, 8$ und dem Knoten 0 sowie eine zusätzliche Vorwärtskante zwischen den Knoten 1 und 2 erzeugt (die rot dargestellten Kanten der ersten Iteration). Die zweite Iteration generiert sogar zweimal die Rückwärtskanten $2 \to 0$ und $6 \to 0$, die mit $1 \to 0$ dazu führen, dass eine dritte Iteration zu ihrer Entfernung durchgeführt werden muss. In CC-MR hängt die Anzahl der durchzuführenden Iterationen nicht nur vom Abstand zwischen den Knoten, sondern auch von den Rückwärtskanten ab, i. e. gäbe es in unserem Beispiel den Knoten 6 nicht, würde wegen der Rückwärtskanten $1 \to 0$ und $2 \to 0$ trotzdem eine dritte Iteration durchgeführt. Die Tabelle 4.1 zeigt die Anzahl der pro Iteration ausgegebenen Vor- und Rückwärtskanten jeder Methode.
Eine weiterführende Verwendung des Knotengrades könnte auch zur Reduzierung der generierten Rückwärtskanten beitragen. Die Hauptaufgabe einer Rückwärtskante im Algorithmus CC-MR ist die Entdeckung von längeren Pfaden in den nächsten Iterationen. Manche dieser Kanten werden jedoch keine längeren Pfade erzeugen, da der Grad ihrer Quellknoten $deg^+ = 1$ ist. Die Rückwärtskanten $4 \to 2$, $6 \to 2$ und $7 \to 2$ in Abbildung 4.11 sind völlig

Iterationen	Berechnung bezüglich der Ordnung „<“		Berechnung bezüglich der Knotengrade	
	Vorwärts-kanten	Rückwärts-kanten	Vorwärts-kanten	Rückwärts-kanten
Iteration 1	9	7	8	5
Iteration 2	10	5	8	0
Iteration 3	8	0	Keine weitere Iteration	

Tabelle 4.1: Anzahl der Vorwärts- und Rückwärtskanten in jeder Iteration bei der Berechnung der zusammenhängenden Komponenten für den Graphen der Abbildung 4.11, links nach der Grundidee von CC-MR und rechts bezüglich der Knotengrade

überflüssig, da ihre Grade $deg^+(4) = deg^+(6) = deg^+(7) = 1$ sind und sie bei der Entdeckung längerer Pfade nicht beteiligt werden.

Für die Umsetzung dieser Ideen werden zwei MapReduce-Jobs benötigt. Der erste Job ist sehr einfach und dient der Berechnung der Grade aller Knoten im Graphen. Der zweite Job verhält sich wie CC-MR. Allerdings werden die Keys und Values mit den im ersten Job berechneten Graden der Knoten angereichert, sodass der Vergleich im CC-MR auf der Basis dieser Grade erfolgt.

5 Auswertung

Zur Evaluierung der Algorithmen, die die transitive Hülle berechnen, wurden die Ergebnisse der Ausführung von Entity Resolution-Workflows auf zwei Datensätze angewandt, *GoogleScholar* und *E-Commerce*. Ein von Dedoop produziertes Match-Ergebnis eines ER-Workflows ist eine auf mehreren Rechnern partitionierte binäre Relation $R(x, y)$ mit der Semantik, dass x und y das gleiche Realweltobjekt beschreiben. Da in beiden Fällen das Match-Ergebnis einem zyklischen Graphen entsprach, wurden lediglich die drei Algorithmen Cyc-Smart-MR, Full-TC-MR und CC-MR hinsichtlich Konvergenz und Laufzeit in zwei verschiedenen Hadoop-Clustern verglichen. Für beide Match-Szenarien wurden ähnliche Verfahren zur Reduktion des Suchraumes, der Ähnlichkeitsberechnung sowie zur Klassifikation verwendet. Die benutzte Blockingstrategie war *Sorted Neighborhood* mit einer Fenstergröße von 100. Als Blockschlüssel wurden dafür normalisierte Attributwertpräfixe der Länge 3 zweier verschiedener Attribute (Titel und Autor bei *GoogleScholar* bzw. Produktbezeichnung und -beschreibung bei *E-Commerce*) generiert. Ähnlichkeiten wurden als Durchschnitt der Trigram-Ähnlichkeiten dieser Attributwerte berechnet. Alle Entitätenpaare mit einer Mindestähnlichkeit von 0.85 wurden zum Match-Ergebnis hinzugefügt. Dieses Setup führt zu einer hohen *precision*, weist wegen der geringen Fenstergröße jedoch einen niedrigen *recall* auf. Mit der Berechnung der transitiven Hülle wird versucht, das Maß *recall* zu verbessern, indem neue matches entdeckt werden. Es ist aber bekannt, dass die Erhöhung des Maßes *recall* eine negative Wirkung auf die *precision* haben könnte. Um dieses Problem zu vermeiden, wird durch das Setzen der Mindestähnlichkeit auf einen hohen Wert (hier 0.85) bei der Ausführung des Match-Verfahrens eine ebenso hohe *precision* nach der Berechnung der transitiven Hülle erzielt.

GoogleScholar enthält 64.263 Publikationsdatensätze. Als Ergebnis des ER-Workflows wurden 3.476 Paare generiert. Die Berechnung der transitiven Hülle erweiterte die Relation um 711 auf 4.187 Paare. Dieses vergleichsweise kleine Problem wurde in einem lokalen Hadoop-Cluster, bestehend aus sechs relativ heterogenen Knoten (Desktop- und Serverrechner mit $2-4$ cores und $4-8$ GB RAM) berechnet. Zur Berechnung wurden 26 map- sowie 26 reduce-Prozesse konfiguriert. Die Laufzeiten und Anzahl der Iterationen jedes Algorithmus sind in der Tabelle 5.1 dargestellt, wobei die jeweils letzte Iteration

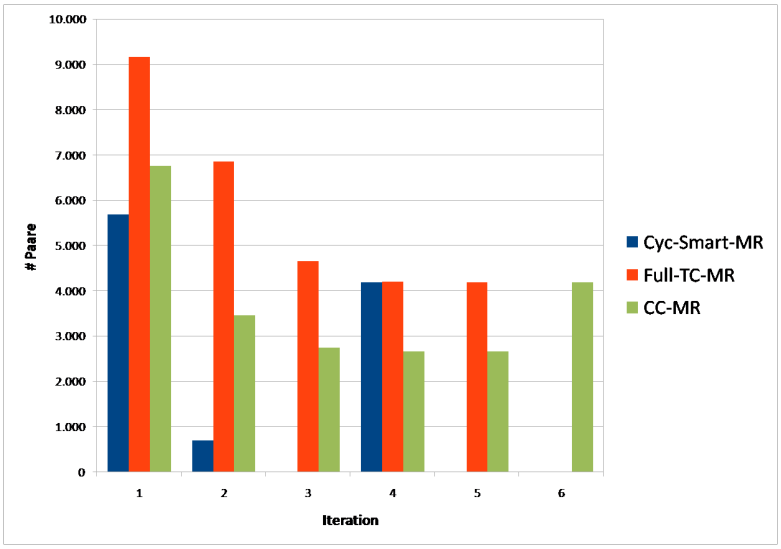

Abbildung 5.1: Datenvolumen von Cyc-Smart-MR, Full-TC-MR und CC-MR in jeder Iteration bei der Berechnung der transitiven Hülle des Datensatzes GoogleScholar

Algorithmen	Iterationen	Laufzeit
Cyc-Smart-MR	4	3 Min. und 29 sec.
Full-TC-MR	5	3 Min. und 50 Sec.
CC-MR	6	4 Min. und 39 Sec.

Tabelle 5.1: Anzahl der Iterationen und Laufzeiten der verschiedenen Algorithmen mit 26 map- und 26 reduce-Tasks für den Datensatz *GoogleScholar*

einen eigenen Job darstellt. Dieser führt die Ergebnisse der vorherigen Iteration zusammen und entfernt die Duplikate im Falle von Cyc-Smart-MR. Bei Full-TC-MR hingegen dient dieser Job lediglich der Entfernung der Duplikate. CC-MR wiederum benötigt diesen Job zur Berechnung der transitiven Hülle der resultierenden zusammenhängenden Komponenten. Die Abbildung 5.1 zeigt die Anzahl der Paare, die die Algorithmen in jeder Iteration generieren. Da die Ausgaben jeder Iteration und die Eingaberelation relativ klein sind, ist die benötigte Zeit zur Verarbeitung der Daten in einer Iteration bei allen Algorithmen nahezu identisch. Aus diesem Grund werden die Laufzeiten im Wesentlichen durch die Anzahl der Iterationen und den damit einhergehenden Overhead eines MapReduce-Jobs (Starten, Einlesen, Umverteilen) dominiert.

E-Commerce umfasst 114.074 Produktbeschreibungen. Insgesamt wurde nach dem Match-Prozess ein Ergebnis von 580.057 Paaren generiert, das sich durch die Berechnung der transitiven Hülle um das 20-fache auf 11.975.615 Paare

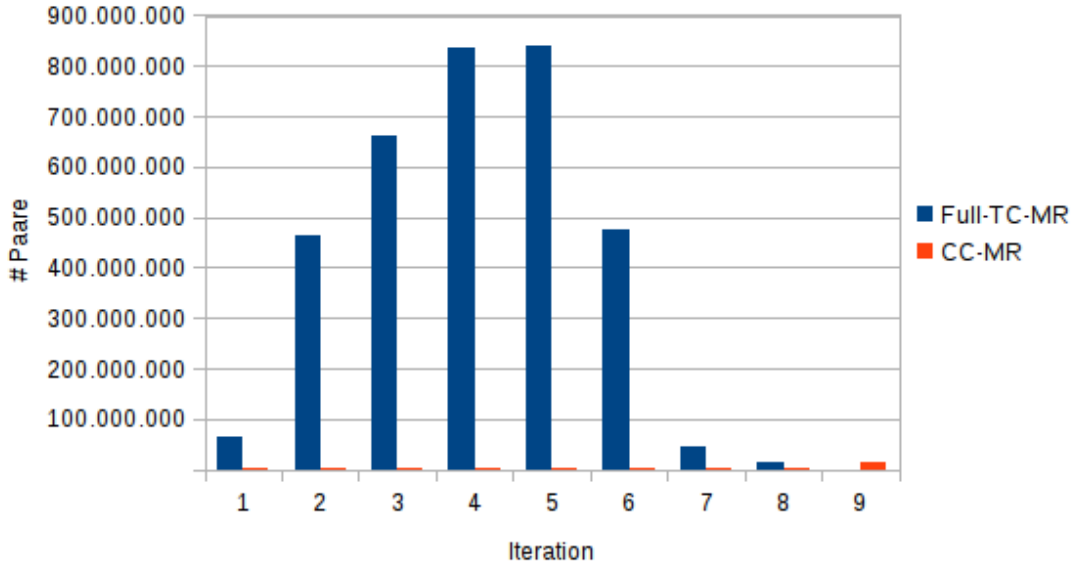

Abbildung 5.2: Datenvolumen von Full-TC-MR und CC-MR in jeder Iteration bei der Berechnung der transitiven Hülle des Datensatzes E-Commerce

Algorithmen	Iterationen	Laufzeit
Cyc-Smart-MR	Out of Memory Exception	
Full-TC-MR	8	33 Min. und 16 Sec.
CC-MR	9	5 Min. und 38 Sec.

Tabelle 5.2: Anzahl der Iterationen und Laufzeiten der verschiedenen Algorithmen mit 150 map- und 150 reduce-Tasks für den Datensatz *E-Commerce*

vergrößerte. Aus diesem Grund wurde ein *Amazon EC2*-Cluster, bestehend aus 50 virtuellen Maschinen des Typs *m1.xlarge* (4 cores und 15 GB RAM), verwendet. Zur Berechnung wurden 150 map- und 150 reduce-Tasks pro Iteration parallel ausgeführt. Die Laufzeiten und die Anzahl der Iterationen jedes Algorithmus sind in Tabelle 5.2 dargestellt. Bei diesem Datensatz werden die Laufzeiten durch die Menge der zu lesenden Daten bestimmt. Im Falle von Cyc-Smart-MR bricht die Ausführung nach der zweiten Iteration mit *Out of Memory Exception* ab. Grund hierfür ist, das der Graph, der *E-Commerce* darstellt, so dicht ist, dass bereits nach der ersten Iteration Keys mit mehr als 190 Values bei demselben reduce-Task gruppiert werden. Ein Key mit n Values bedeutet für Cyc-Smart-MR $\frac{n \times (n-1)}{2}$ Paare mit ihren jeweiligen Zwischenknoten einzulesen und zusammen im Hauptspeicher zu bearbeiten. Beispielsweise werden aus einem Key mit 190 Values 17.955 Paare mit ihren jeweiligen

Zwischenknoten generiert, die im Hauptspeicher gehalten werden müssen. Das Problem hier ist nicht die Anzahl der Paare, sondern es sind ihre zum Teil redundanten Zwischenknoten. Zum Beheben des Problems wurde Full-TC-MR, der keine Zwischenknoten speichert, präsentiert. Full-TC-MR erkennt zwar in Iteration i Duplikate, die in vorherigen Iterationen erzeugt wurden, jedoch keine, die in der Iteration i selbst erzeugt werden. Diese werden erst in der Iteration $i + 1$ beseitigt. Die auf diese Weise entstandenen Duplikate lassen die Datenmenge rasch ansteigen. Wie die Abbildung 5.2 deutlich zeigt, erreicht die Ausgabe der fünften Iteration beispielsweise $840.589.523$ Paare, deren Bearbeitung viel Zeit in Anspruch nimmt. Die Arbeitsweise von CC-MR bei der Entdeckung zusammenhängender Komponenten erzeugt kleinere Ausgaben mit nahezu gleicher Anzahl von Paaren, wie in der Abbildung 5.2 zu sehen ist. In jeder Iteration, außer der letzten, muss CC-MR für einen Key, der n Values besitzt, nur diese n Values mit ihrem Key im Hauptspeicher halten. Falls der Key die kleinste Id besitzt, werden lediglich n Paare generiert, andernfalls maximal $2 \times (n + 1)$ Paare, welche die Vor- und Rückwärtskanten darstellen. Demzufolge benötigt er trotz der höheren Anzahl von Iterationen im Vergleich zu Full-TC-MR deutlich weniger Zeit (siehe Tabelle 5.2).
Besteht jedoch der Ausgangsgraph aus einer sehr großen zusammenhängenden Komponente und anderen viel kleineren, so könnte die letzte Iteration von CC-MR, bei der ein kartesisches Produkt der Knoten berechnet wird, einen Nachteil aufweisen. Der reduce-Task, der die transitive Hülle der größten zusammenhängenden Komponente berechnet, wird immer noch Paare generieren und in das Dateisystem HDFS schreiben, während die anderen reduce-Tasks, die die kleineren zusammenhängenden Komponenten bearbeiten, längst fertig sind. Da ein MapReduce-Job erst endet, wenn alle gestarteten reduce-Tasks beendet sind, wird die Laufzeit des Algorithmus von dem reduce-Task dominiert, der die größte zusammenhängende Komponente bearbeitet. Das Problem, auch unter dem Namen *Data Skew* bekannt, ist im Falle der Berechnung der transitiven Hülle schwer zu handhaben, da für die Realisierung des kartesischen Produktes alle Entitäten auf einem Rechner liegen müssen.

Darüber hinaus existiert zwischen CC-MR und den anderen Algorithmen (Cyc-Smart-MR und Full-TC-MR) ein Unterschied hinsichtlich der Pfadlänge. In CC-MR kann die Pfadlänge nicht ermittelt werden, da der komplette Graph verändert wird. In den beiden anderen Algorithmen jedoch wird die Pfadlänge stets zusammen mit dem Pfad gespeichert. Diese Eigenschaft kann beispielsweise bei der Begrenzung der Anzahl der durchzuführenden Iterationen oder bei der Auswahl der auszugebenden Pfade verwendet werden. So ist es möglich, die Berechnung der transitiven Hülle zu terminieren, wenn irgendein Pfad eine zuvor bestimmte Länge erreicht hat. Es ist insofern wichtig, als lange Pfade in der transitiven Hülle eines Match-Ergebnisses zu schlechter *precision*

führen können (chain problem) [Chr12]. Obwohl in dieser Arbeit die Qualität der Ergebnisse nach der Berechnung der transitiven Hülle nicht geprüft wurde, kann das Problem mit den in Tabellen 5.1 und 5.2 dargestellten Ergebnissen erläutert werden. Die Tabelle 5.1 zeigt, dass Cyc-Smart-MR die transitive Hülle in vier Iterationen berechnet, wobei die letzte Iteration nur eine duplikatfreie Union der Ergebnisse durchführt. Daraus ergibt sich, dass der maximale Abstand zwischen zwei Entitäten die Länge 7 hat. Die langen Pfade (z. B. länger als 5) werden wahrscheinlich einen negativen Einfluss auf die *precision* haben. Da aber die Menge der neu generierten Paare relativ gering ist (711 neue Paare), würden diese langen Pfade die Qualität des Ergebnisses vermutlich nicht so stark beeinträchtigen. Bei der Berechnung der transitiven Hülle für das Match-Ergebnis des Datensatzes *E-Commerce* liefern die Ergebnisse jedoch ein anderes Bild. Hier gehen wir davon aus, dass die Anzahl der tatsächlich benötigten Iterationen 6 ist. Das bedeutet, dass Pfade mit einer Länge von bis zu 63 generiert wurden (Pfade mit Längen über 33 wurden tatsächlich entdeckt). Für ein Match-Ergebnis könnte dies zu einer schlechten *precision* führen, zumal das Ergebnis der Berechnung der transitiven Hülle die Ausgangsrelation um das 20-fache erweitert hat. Sind aber die Längen der Pfade bekannt, kann innerhalb einer beliebigen Iteration die Generierung der Pfade eingeschränkt werden, sodass nur Pfade ausgegeben werden, deren Längen kleiner als ein festgelegter Wert sind. Diese Lösung ist bei Cyc-Smart-MR und Full-TC-MR umsetzbar, jedoch nicht bei CC-MR.

6 Zusammenfassung und Ausblick

Die vorliegende Arbeit hat sich das Ziel gesetzt, eine Methode zur Berechnung der transitiven Hülle eines Match-Ergebnisses mit MapReduce zu implementieren. Diese sollte in das Tool Dedoop als abschließende Phase eines Entity-Resolution-Workflows integriert werden, um die Qualität des Match-Ergebnisses zu verbessern. Zu diesem Zweck wurden unterschiedliche iterative Algorithmen für die Berechnung der transitiven Hülle eines Graphen untersucht, wobei den parallelen Algorithmen eine besondere Bedeutung zukam. Die Untersuchungen zeigten, dass bei der iterativen Berechnung der transitiven Hülle zwei Kriterien entscheidend sind, nämlich die Anzahl der benötigten Iterationen zum Konvergieren und die Menge der generierten Duplikate pro Iteration. Für einen azyklischen Graphen der Tiefe d haben manche Algorithmen eine $\mathcal{O}(log\, d)$ Laufzeitkomplexität, aber sie generieren eine Vielzahl von Duplikaten, deren Entfernung in einem verteilten System einen erheblichen Aufwand verursacht. Andere Algorithmen hingegen generieren bei solchen Graphen keine Duplikate, haben aber eine $\mathcal{O}(d)$ Laufzeitkomplexität.

Eine mögliche MapReduce-Implementierung des bekanntesten Verfahrens aus der Literatur, Smart, wurde zwar vorgeschlagen, jedoch verwendet diese für die Entfernung von Duplikaten zwei teure algebraische Operationen, Union und Differenz. In einem Hadoop-Cluster führen diese zusätzlichen Operationen zu einer Verdreifachung der Anzahl der zum Konvergieren benötigten Iterationen. In dieser Arbeit wurde die Grundidee der Berechnung der transitiven Hülle eines azyklischen Graphen mit MapReduce präsentiert, die aufgrund ihres logarithmischen Verhaltens ebenfalls Duplikate generiert. Auf die Verwendung von algebraischen Operationen für die Entfernung von Duplikaten wurde hier verzichtet. Stattdessen lag der Fokus auf der Analyse und Beseitigung der Ursachen für die Generierung von Duplikaten. Daraus resultierend wurden drei Kriterien zur Vermeidung der Erzeugung von Duplikaten bei der Berechnung der transitiven Hülle eines azyklischen Graphen definiert. Außerdem wurde eine neue Methode zur Terminierung präsentiert, die nicht auf der Generierung von neuen Paaren, sondern auf der Pfadlänge beruht. Der Algorithmus Smart-MR verwendet diese Kriterien und berechnet die transitive Hülle eines azyklischen Graphen in genau $\log\, d$

Iterationen. Für ungerichtete zyklische Graphen wie das Match-Ergebnis wurden Cyc-Smart-MR und Full-TC-MR, zwei Varianten von Smart-MR implementiert. Außerdem wurde ein bereits existierender Algorithmus, CC-MR, der zusammenhängende Komponenten eines Graphen entdeckt, für die Berechnung der transitiven Hülle leicht angepasst. Die drei zuletzt genannten Algorithmen weisen ebenfalls ein logarithmisches Verhalten auf.

Die Experimente verdeutlichten das Problem des Speicherbedarfs von Cyc-Smart-MR. Bei dichten Graphen bricht seine Ausführung nach wenigen Iterationen ab. Full-TC-MR erzeugt pro Iteration eine große Datenmenge, deren Bearbeitung viel Zeit in Anspruch nimmt. Die Arbeitsweise von CC-MR hingegen führt zur Generierung von kleineren Datenmengen. Aus diesem Grund zeigt CC-MR im Vergleich zu Full-TC-MR eine bessere Performance, besonders bei großen Datenmengen.

Trotz der guten Ergebnisse von CC-MR hat seine Grundidee einige Schwächen, die zur Generierung unnötiger Daten führen. Diese wurden in Abschnitt 4.2.5 diskutiert und sind Gegenstand zukünftiger Arbeiten. Außerdem wäre es sinnvoll, die Algorithmen an gerichtete Graphen anzupassen und zu evaluieren.

Nach meinem Kenntnisstand präsentiert diese Arbeit die ersten praktischen Implementierungen der transitiven Hülle in MapReduce für das Entity-Resolution-Problem. Die Ergebnisse lassen sich jedoch problemlos auf andere Domänen übertragen.

Literaturverzeichnis

[ABC+11] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, pages 1–8, 2011. 31, 36

[AHB86] Peter M. G. Apers, Maurice A. W. Houtsma, and Frank Brandse. Processing recursive queries in relational algebra. In *DS-2'86*, pages 17–39, 1986. 25

[Ban85] F. Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems (Islamorada)*, pages 165–178, New York, NY, USA, 1985. Springer-Verlag New York, Inc. 23, 27

[BEH+10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010. 33

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM. 24

[CCH93] F. Cacace, S. Ceri, and M.A.W. Houtsma. A survey of parallel execution strategies for transitive closure and logic programs. *Int. Journal of Distributed and Parallel Databases*, 1(4):337–382, October 1993. Imported from EWI/DB PMS [db-utwente:arti:0000002031]. 22, 27

[CdM89] Jean-Pierre Cheiney and Christophe de Maindreville. A parallel transitive closure algorithm using hash-based clustering. In Haran Boral and Pascal Faudemay, editors, *IWDM*, volume 368 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 1989. 28

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on*

Knowl. and Data Eng., 1(1):146–166, March 1989. 19

[Chr12] Peter Christen. *Data Matching, Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer-Verlag Berlin Heidelberg 2012, 2012. 1, 10, 21, 63

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. 1

[ES07] Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007. 11

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. 1

[GM89] John Grant and Jack Minker. Deductive database theories. *The Knowledge Engineering Review*, 4:267–304, 12 1989. 19

[Had] Hadoop homepage. `http://hadoop.apache.org/`. 1

[Ioa86] Yannis E. Ioannidis. On the computation of the transitive closure of relational operators. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 403–411, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc. 26

[KR13] Lars Kolb and Erhard Rahm. Parallel Entity Resolution with Dedoop. *Datenbank-Spektrum*, 13(1):23–32, 2013. III, 12, 15

[KTR12a] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient Deduplication with Hadoop. *PVLDB*, 5(12):1878–1881, 2012. 2

[KTR12b] Lars Kolb, Andreas Thor, and Erhard Rahm. Multi-pass sorted neighborhood blocking with MapReduce. *Computer Science - R&D*, 27(1):45–63, 2012. 2

[SBF12] Thomas Seidl, Brigitte Boden, and Sergej Fries. Cc-mr - finding connected components in huge graphs with mapreduce. In *ECML/PKDD (1)*, pages 458–473, 2012. 52

[TMC+08] Tran Doan Thanh, Subaji Mohan, Eunmi Choi, SangBum Kim, and Pilsung Kim. A taxonomy and survey on distributed file systems. In *Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management - Volume*

01, NCM '08, pages 144–149, Washington, DC, USA, 2008. IEEE Computer Society. 4

[UGMW01] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. 19

[VK88] Patrick Valduriez and Setrag Khoshafian. Parallel evaluation of the transitive closure of a database relation. *Int. J. Parallel Program.*, 17(1):19–42, February 1988. 27

[War62] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962. 22

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Ort Datum Unterschrift